Game development in Rust with Macroquad

Olle Wreede

Game development in Rust with Macroquad

Copyright © 2025 by Olle Wreede

Licensed under CC BY-SA 4.0 https://creativecommons.org/licenses/by-sa/4.0/ (https://creativecommons.org/licenses/by-sa/4.0/)

This license enables reusers to distribute, remix, adapt, and build upon the material in any medium or format, so long as attribution is given to the creator. The license allows for commercial use. If you remix, adapt, or build upon the material, you must license the modified material under identical terms. CC BY-SA includes the following elements:

BY: credit must be given to the creator.
SA: Adaptations must be shared under the same terms.

First print edition February 2025

https://mq.agical.se (https://mq.agical.se)

Proofreading and consistency check: Kathrin Hagmaier

Additional content by Peter Strömberg

Cover design by Niklas Elmgren

Ferris the Teacher (https://www.behance.net/gallery/89117181/Ferris-the-professional) image by Esther Arzola (https://www.behance.net/estherarzola) is licensed under CC BY 4.0 (https://creativecommons.org/licenses/by/4.0/deed.en).

ISBN: 978-91-976896-0-1

Acknowledgments

Thanks to mom and dad, and the rest of my family. I want to give an extra special thanks to the following people:

- Niklas Elmgren for designing the cover
- Kathrin Hagmaier for help with proofreading and consistency check
- Peter Strömberg for user testing and additional content
- The Art of Code for the starfield shader
- Esther Arzola for the beautiful image of Ferris with a teacher's hat
- Fedor Logachev for developing Macroquad
- Everyone who designed game assets and placed them in the public domain

And of course to everybody who has followed the guide, has cheered me on or given feedback and suggestions.

Table of contents

Copyright
Acknowledgments
Table of contents
Game development in Rust with Macroquad

- Your first Macroquad app 8
- Fly away 14
- Smooth movement 18
- Falling squares 21
- Collision 26
- Bullet hell 31
- Points 36
- Game state 40
- Starfield shader 47
- Particle explosions 53
- Graphics 58
  - Spaceship and bullets
  - Graphical explosions
  - Animated enemies
- Music and sound effects 74
- Graphical menu 77
- Resources 83
  - Resources and errors
  - Coroutines and Storage
- Release your game 95
  - Build for desktop
  - Build for the web
  - Build for Android
  - Build for iOS
- The end 108

Full source code
Credits Glossary

Game development in Rust with Macroquad

This guide is written by Olle Wreede at Agical (https://www.agical.se/).

Olle Wreede is a senior software developer working at Agical where he helps customers build better software. He has worked in the software industry for two and a half decades.

Agical helps their customers deliver software earlier. By focusing on helping everyone involved, getting better at working together and improving their technical skills.

This guide is available online at the following address: https://mq.agical.se/

> **Info**
>
> The source code for all chapters of this book is available here:
> https://mq.agical.se/github.html (https://mq.agical.se/github.html)

Game development guide

In this guide we will be developing a game from scratch. In each chapter we will add a small feature to the game that explains a part of the Macroquad library. In the beginning the game will be very simple, but at the end of the guide you will have built a complete game with graphics and sound. You will be able to build the game for desktop computers, the web, as well as mobile devices.

The game we are making is a classic shoot 'em up where the player controls a spaceship that has to shoot down enemies flying down from the top of the screen.

> **Challenge**
>
>
>
> This is Ferris, the teacher who will show up at the end of every chapter to give you an extra challenge. Doing the challenge is optional; you can continue to the next chapter without it.

The Macroquad game library

Macroquad is a game library for the programming language Rust. It includes everything you need to develop a 2D game. The main advantage of Macroquad compared with other game libraries is that it works with many different platforms. Since it has very few dependencies it also compiles very fast.

With Macroquad it's possible to develop games for desktop operating systems like Windows, Mac, and Linux. It also has support to compile for mobile devices like iOS and Android. Thanks to the WebAssembly support it can also be compiled to run in a web browser. All this can be done without having to write any platform specific code.

The library has efficient 2D rendering support, and some rudimentary 3D features. It also includes a simple immediate UI library to make graphical game interfaces.

This guide assumes some prior knowledge of Rust programming. However there are no advanced concepts, so if you already know another programming language it is still possible to follow along. More information about Rust is available in the Rust book (https://doc.rust-lang.org/book/) that is available online. I can also recommend the book Hands-on Rust (https://hands-on-rust.com/) by Herbert Wolverson where you learn Rust by writing a roguelike game.

Find more info on the Macroquad homepage

On the Macroquad homepage (https://macroquad.rs) there are examples (https://macroquad.rs/examples/) of how different features of Macroquad work, Macroquad-related articles (https://macroquad.rs/articles), and documentation of the API (https://docs.rs/macroquad/latest/macroquad/).

Macroquad version 0.4

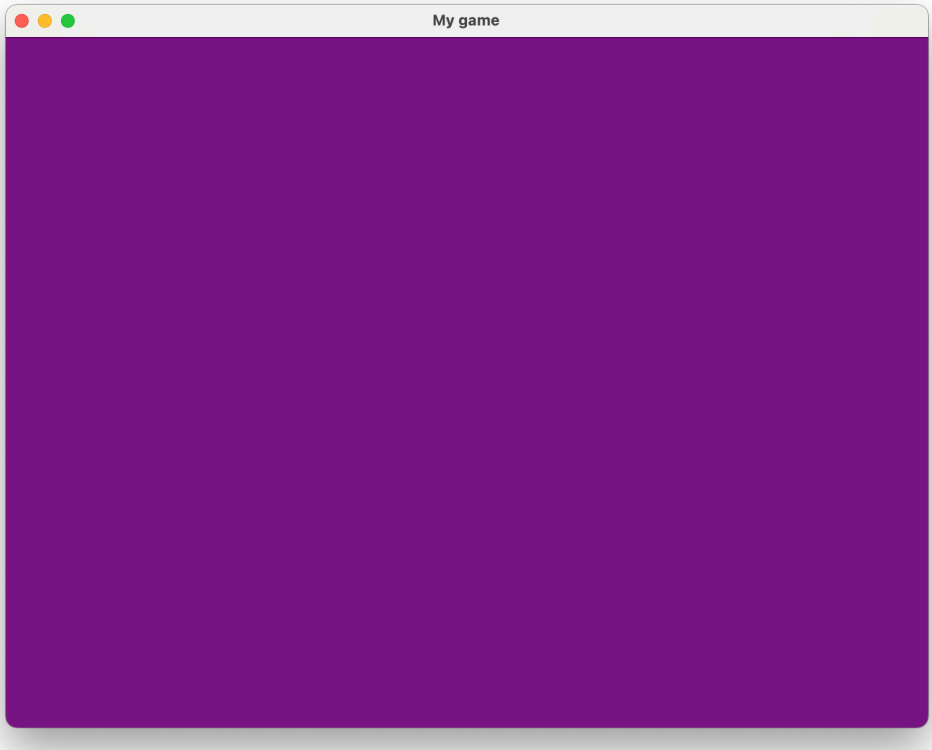
This guide is written for version 0.4 of Macroquad. It may not work for future versions because Macroquad is under active development.

PDF book

This guide is also available as a downloadable PDF book (https://mq.agical.se/pdf/output.pdf).

Game development in Rust with Macroquad (https://mq.agical.se/) by Olle Wreede (https://olle.wreede.se/) is licensed under
CC BY-SA 4.0 (https://creativecommons.org/licenses/by-sa/4.0/?ref=chooser-v1)

Your first Macroquad app



Now it's time to develop your first application with Macroquad. Start by installing (https://www.rust-lang.org/learn/get-started) the programming language Rust if you don't already have it.

Implementation

Create a new Rust project using the Cargo command line tool and add `macroquad` with version `0.4` as a dependency. If you want, you can give your game a more interesting name than "my-game".

```
cargo new --bin my-game
cd my-game/
cargo add macroquad@0.4
```

Your `Cargo.toml` file should now look like this:

```
[package]
name = "my-game"
version = "0.1.0"
edition = "2024"

# See more keys and their definitions at https://doc.rust-
lang.org/cargo/reference/manifest.html

[dependencies]
macroquad = "0.4"
```

Open the file `src/main.rs` in your favorite text editor and change the content to look like this:

```
use macroquad::prelude::*;

#[macroquad::main("My game")]
async fn main() {
    loop {
        clear_background(DARKPURPLE);
        next_frame().await
    }
}
```

Run your application with `cargo run`, and a new window with a dark purple background will open once the compilation has finished.

Description of the application

The first line is used to import everything you need from Macroquad. This is most easily done by importing `macroquad::prelude::*`, but it is also possible to import only the features that are used.

The attribute `#[macroquad::main("My game")]` is used to tell Macroquad which function will be run when the application starts. When the application is started, a window will open with the argument as the title, and the function will be executed asynchronously. If you have named your game something more interesting you should change the text `My game´ to the name of your game.

> **Control the window configuration**
>
> To change the configuration for the window, such as the size or whether it should start in fullscreen mode, you can use the struct Conf (https://docs.rs/macroquad/latest/macroquad/window/struct.Conf.html) instead of the string as an argument.

Inside the `main` function there is a loop that never ends. All the game logic will be placed inside this game loop and will be executed in every frame. In our case we clear the background of the window

with a dark purple color with the function `clear_background(DARKPURPLE)`. At the end of the loop is the function `next_frame().await` which will wait until the next frame is available.

> **Clears with black as default**
>
> Even if `clear_background()` isn't used explicitly, the screen will be cleared with a black color at the start of each frame.

> **Challenge: Change background color**
>
>
>
> Try changing the background of the window to your favorite color.

Publish on the web (if you want)

One of the big advantages with Rust and Macroquad is that it is very easy to compile a standalone application for different platforms. How this works will be explained in a later chapter of this guide. If you want, you can setup a GitHub deploy action to publish a web version of the game every time you commit.

When you created the game with `cargo new` a local Git repository was also created. Start by committing your changes locally. After that you can create a repository on GitHub and push the code there.

> **The name of the game**
>
> The two files below refer to `my-game.wasm`. If you've changed the name of your crate to something other than `my-game` you need to change those references.

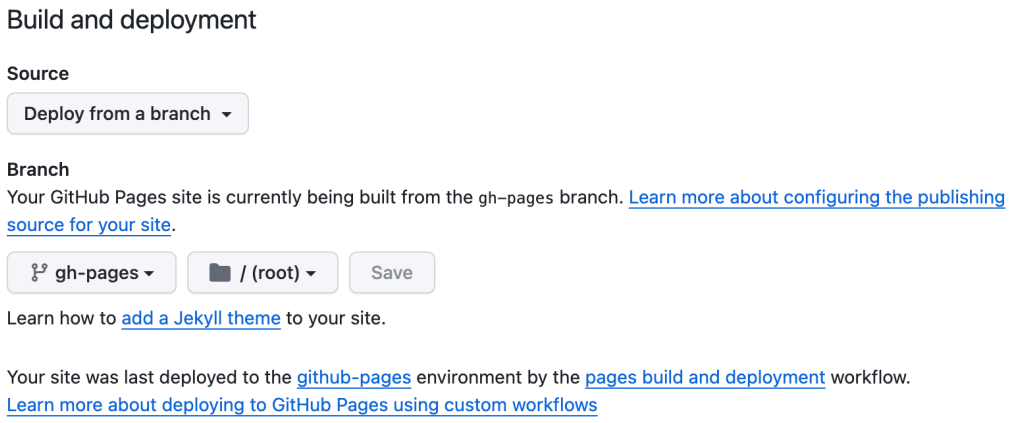
You need an HTML file to show the game. Create a file called `index.html` in the root of the project/crate and add the following content:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1" />
    <title>My Game</title>
    <style>
        html,
        body,
        canvas {
            margin: 0;
            padding: 0;
            width: 100%;
            height: 100%;
            overflow: hidden;
            position: absolute;
            background: black;
            z-index: 0;
        }
    </style>
</head>
<body>
    <canvas id="glcanvas" tabindex='1'></canvas>
    <!-- Minified and statically hosted version of https://github.com/not-
fl3/macroquad/blob/master/js/mq_js_bundle.js -->
    <script src="mq_js_bundle.js"></script>
    <script>load("my-game.wasm");</script> <!-- Your compiled WASM binary -->
</body>
</html>
```

The following GitHub Actions Workflow will compile the game to WASM and put all files in place so that the game will work on the web. Place the code in `.github/workflows/deploy.yml`.

```yaml
name: Build and Deploy
on:
  push:
    branches:
      - main # If your default branch is named something else, change this

permissions:
  contents: write
  pages: write

jobs:
  build-and-deploy:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout
        uses: actions/checkout@v2

      - name: Install Rust
        uses: actions-rs/toolchain@v1
        with:
          toolchain: stable
          target: wasm32-unknown-unknown
          override: true

      - name: Build
        run: cargo build --release --target wasm32-unknown-unknown

      - name: Prepare Deployment Directory
        run: |
          mkdir -p ./deploy
          cp ./target/wasm32-unknown-unknown/release/my-game.wasm ./deploy/
          cp index.html ./deploy/

      - name: Deploy
        uses: peaceiris/actions-gh-pages@v3
        with:
          github_token: ${{ secrets.GITHUB_TOKEN }}
          publish_dir: ./deploy
```

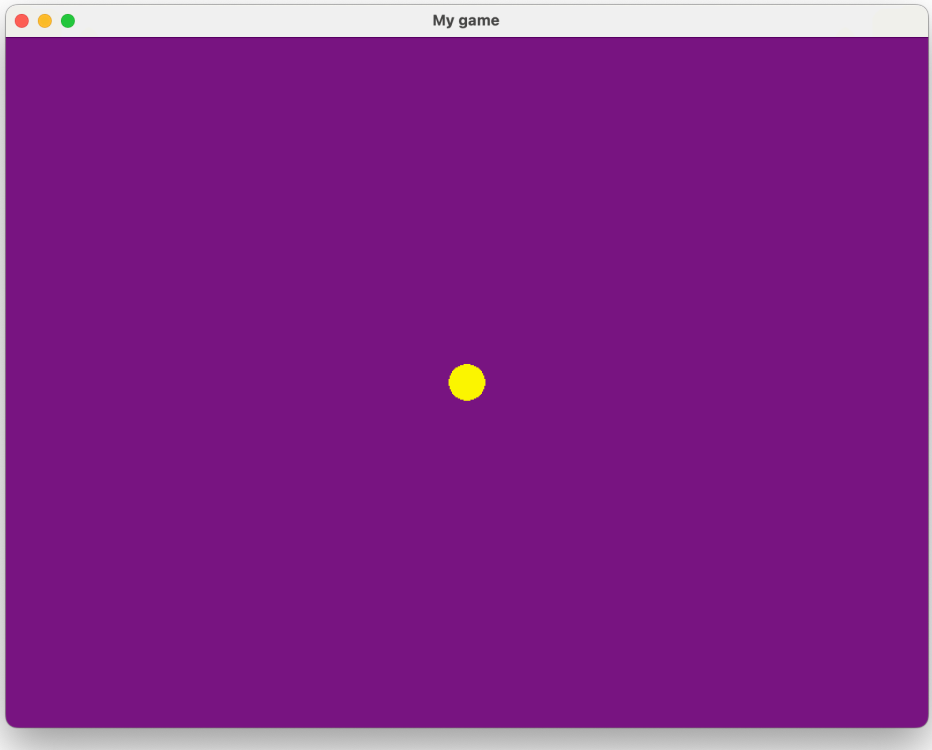
Commit and push! You can follow the build under the **Actions** page of the repository. The first time you push your code the game will be built and all files placed in the correct place, in the root of the branch `gh-pages`, but no web page will be created. You need to change a configuration of the GitHub repository under **Settings** > **Pages** > **Build and deployment**. Set `gh-pages` as the branch from which to deploy the web page.


Build and deployment
Source
Deploy from a branch
Branch
Your GitHub Pages site is currently being built from the gh-pages branch. Learn more about configuring the publishing source for your site.
gh-pages
/ (root)
Save
Learn how to add a Jekyll theme to your site.
Your site was last deployed to the github-pages environment by the pages build and deployment workflow.
Learn more about deploying to GitHub Pages using custom workflows

When the build is done you will be able to play your game on `https://<your-github-account>.github.io/<repository-name>`.

It won't be much of game yet, only a purple background. But you have delivered early, and the project is configured for continuous delivery. Every time you add functionality to the game and push the code to GitHub, you will be able to play the latest version of the game on the web. In the next chapter things will start to move!

Fly away



A game is not much fun without something happening on the screen. To begin with, we will show a circle that we can steer with the arrow keys on the keyboard.

Implementation

The first two lines of the `main` function uses the functions `screen_width()` and `screen_height()` to get the width and height of the application window. These values are divided by `2` to get the coordinates of the center of the window, and stored in the variables `x` and `y`. These variables will be used to decide where to draw the circle on the screen.

```
let mut x = screen_width() / 2.0;
let mut y = screen_height() / 2.0;
```

Handle keyboard input

Inside the main loop we will still clear the background as it should be done in each frame. After that there are four `if` statements that check if any of the arrow keys on the keyboard has been pressed. The variables `x` and `y` are changed to move the circle in the corresponding direction.

The function `is_key_down()` returns true if the given key is being pressed during the current frame. The argument is of the enum `KeyCode` that contains all keys available on the keyboard.

> **More on enums**
>
> You can read more about how the Rust `enum` feature (https://doc.rust-lang.org/book/ch06-00-enums.html) works in the Rust book.

```
if is_key_down(KeyCode::Right) {
    x += 1.0;
}
if is_key_down(KeyCode::Left) {
    x -= 1.0;
}
if is_key_down(KeyCode::Down) {
    y += 1.0;
}
if is_key_down(KeyCode::Up) {
    y -= 1.0;
}
```

> **More keycodes!**
>
> You can find other available keys in the documentation of KeyCode (https://docs.rs/macroquad/latest/macroquad/input/enum.KeyCode.html).

Draw a circle

Finally we will draw a circle on the screen at the coordinates in `x` and `y`. The circle has a radius of 16 and will be drawn in a yellow color.

```
draw_circle(x, y, 16.0, YELLOW);
```

Colors and shapes

Macroquad has several constants for common colors (https://docs.rs/macroquad/latest/macroquad/color/colors/index.html), and you can also use the macro `color_u8` (https://docs.rs/macroquad/latest/macroquad/macro.color_u8.html) to create a color with specific values for red, green, blue, and transparency.

The other shapes that can be drawn with Macroquad are described in the documentation of Macroquad's Shape API (https://docs.rs/macroquad/latest/macroquad/shapes/index.html).

Challenge: Faster movement

Change the value added to `x` and `y` to define how fast the circle will move.

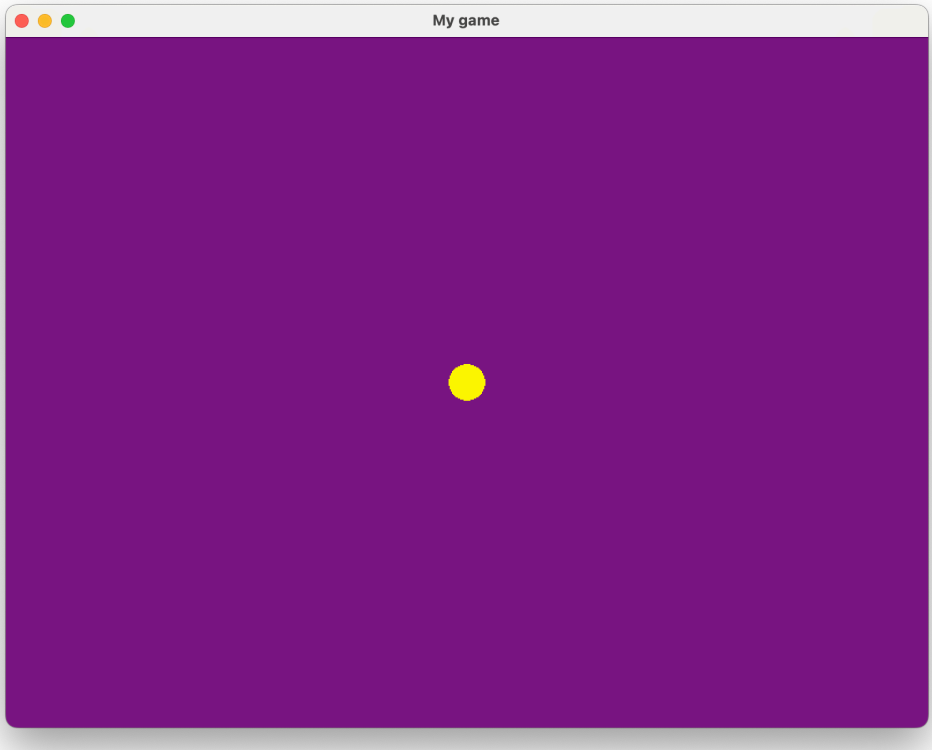
Source

The source of `main.rs` should look like this:

```
use macroquad::prelude::*;

#[macroquad::main("My game")]
async fn main() {
    let mut x = screen_width() / 2.0;
    let mut y = screen_height() / 2.0;

    loop {
        clear_background(DARKPURPLE);

        if is_key_down(KeyCode::Right) {
            x += 1.0;
        }
        if is_key_down(KeyCode::Left) {
            x -= 1.0;
        }
        if is_key_down(KeyCode::Down) {
            y += 1.0;
        }
        if is_key_down(KeyCode::Up) {
            y -= 1.0;
        }

        draw_circle(x, y, 16.0, YELLOW);

        next_frame().await
    }
}
```

When you run the game, a yellow circle will appear in the middle of the window. Try using the arrow keys to move the circle around.

Smooth movement



Since Macroquad will draw frames as quickly as possible, we need to check how much time has passed between each update to determine how far the circle should move. Otherwise, our game will run at different speeds on different computers, depending on how quickly they can run the application. The specific framerate will depend on your computer; if Vsync is enabled it may be locked to 30 or 60 frames per second.

Implementation

We will expand the application and add a constant that determines how quickly the circle should move. We call the constant `MOVEMENT_SPEED` and assign the value `200.0`. If the circle moves too fast or too slow, we can decrease or increase this value.

```
const MOVEMENT_SPEED: f32 = 200.0;
```

Time between frames

Now we will use the function `get_frame_time()` to get the time in seconds that has passed since the last frame. We assign this value to a variable called `delta_time` that we will use later.

```
let delta_time = get_frame_time();
```

Update movement

When the variables `x` and `y` are updated, we will multiply the values of the constant `MOVEMENT_SPEED` by the variable `delta_time` to get how far the circle should move during this frame.

```
if is_key_down(KeyCode::Right) {
    x += MOVEMENT_SPEED * delta_time;
}
if is_key_down(KeyCode::Left) {
    x -= MOVEMENT_SPEED * delta_time;
}
if is_key_down(KeyCode::Down) {
    y += MOVEMENT_SPEED * delta_time;
}
if is_key_down(KeyCode::Up) {
    y -= MOVEMENT_SPEED * delta_time;
}
```

Limit movement

Finally, we will prevent the circle from moving outside of the window. We use the Macroquad function `clamp()` to make sure `x` and `y` are never below `0` or above the width of the window.

```
x = clamp(x, 0.0, screen_width());
y = clamp(y, 0.0, screen_height());
```

> **The Macroquad Math API**
>
> The `clamp()` function is used to clamp a value between a minimum and maximum value. It is part of the Macroquad Math API (https://docs.rs/macroquad/latest/macroquad/math/index.html).

Challenge: Clamp movement

Change the constant `MOVEMENT_SPEED` if the circle is moving too slow or too fast.

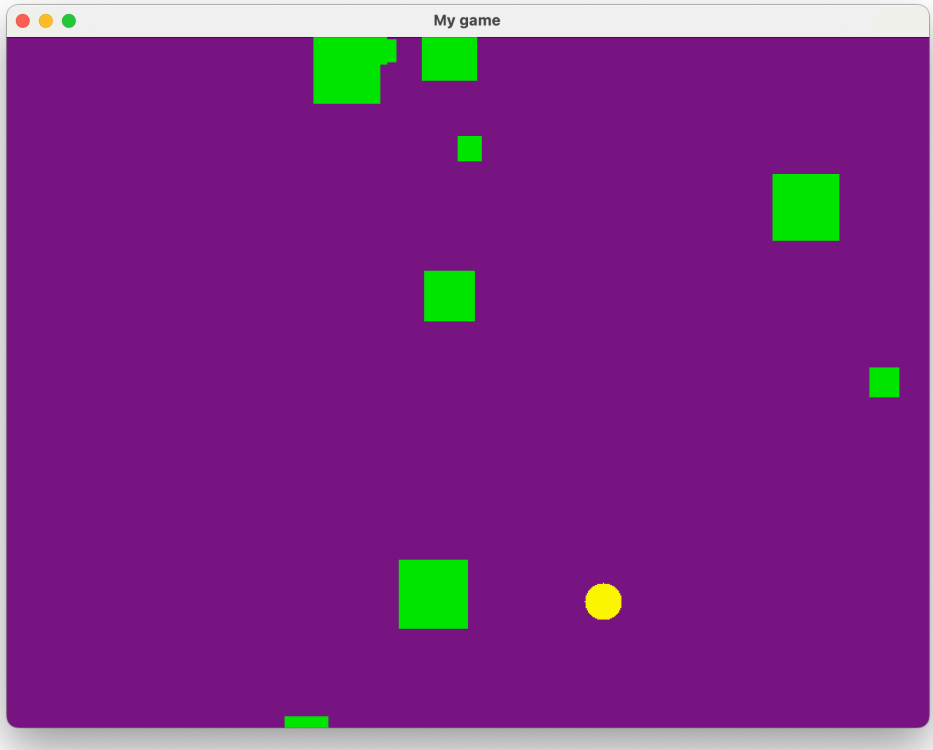
What do you need to change to ensure that the entire circle stays within the window when the position is clamped?

Source

The code should now look like this:

```
use macroquad::prelude::*;

#[macroquad::main("My game")]
async fn main() {
    const MOVEMENT_SPEED: f32 = 200.0;

    let mut x = screen_width() / 2.0;
    let mut y = screen_height() / 2.0;

    loop {
        clear_background(DARKPURPLE);

        let delta_time = get_frame_time();
        if is_key_down(KeyCode::Right) {
            x += MOVEMENT_SPEED * delta_time;
        }
        if is_key_down(KeyCode::Left) {
            x -= MOVEMENT_SPEED * delta_time;
        }
        if is_key_down(KeyCode::Down) {
            y += MOVEMENT_SPEED * delta_time;
        }
        if is_key_down(KeyCode::Up) {
            y -= MOVEMENT_SPEED * delta_time;
        }

        x = clamp(x, 0.0, screen_width());
        y = clamp(y, 0.0, screen_height());

        draw_circle(x, y, 16.0, YELLOW);
        next_frame().await
    }
}
```

Falling squares



To make sure there is something happening in our game, it's time to create some action. Since the hero in our game is a brave circle, our opponents will be squares falling down from the top of the window.

Implementation

Struct for shapes

To keep track of our circle and all the squares, we'll create a struct that we can name `Shape`, which will contain the size and speed, as well as `x` and `y` coordinates.

```
struct Shape {
    size: f32,
    speed: f32,
    x: f32,
    y: f32,
}
```

Initialize random number generator

We'll use a random number generator to determine when new squares should appear on the screen, how big they should be and how fast they will move. Therefore, we need to seed the random generator so that it doesn't produce the same random numbers every time. This is done at the beginning of the `main` function using the `rand::srand()` method, to which we pass the current time as the seed.

```
    rand::srand(miniquad::date::now() as u64);
```

> **Miniquad in play!**
>
> We are using the function `miniquad::date::now()` from the graphics library Miniquad (https://docs.rs/miniquad/latest/miniquad/index.html) to get the current time.

Vector of squares

At the beginning of the `main` function we create a vector called `squares` that will contain all the squares to be displayed on the screen. The new variable `circle` will represent our hero, the amazing circle. The speed uses the constant `MOVEMENT_SPEED`, and the `x` and `y` fields are set to the center of the screen.

```
    let mut squares = vec![];
    let mut circle = Shape {
        size: 32.0,
        speed: MOVEMENT_SPEED,
        x: screen_width() / 2.0,
        y: screen_height() / 2.0,
    };
```

Start by modifying the program so that `circle` is used instead of the variables `x` and `y` and confirm that everything works as it did before adding the enemy squares.

> **Getting a "type annotation needed" warning?**
>
> The Rust compiler might warn about "type annotations needed" on the Vector. Once we add an enemy square in the next section that warning should disappear.

Add enemy squares

It's time to start the invasion of evil squares. Here, just like before, we split updating the movement and drawing the squares. This way, the movement does not depend on the screen's refresh rate, ensuring that all changes are done before we start drawing anything to the screen.

First, we use the function `rand::gen_range()` to determine whether to add a new square. It takes two arguments, a minimum value and a maximum value, and returns a random number between those two values. We generate a random number between 0 and 99, and if the value is 95 or higher, a new `Shape` is created and added to the `squares` vector. To add some variation, we also use `rand::gen_range()` to get different size, speed, and starting position of every square.

```
if rand::gen_range(0, 99) >= 95 {
    let size = rand::gen_range(16.0, 64.0);
    squares.push(Shape {
        size,
        speed: rand::gen_range(50.0, 150.0),
        x: rand::gen_range(size / 2.0, screen_width() - size / 2.0),
        y: -size,
    });
}
```

> **The coordinate system starts top left**
>
> Rectangles are drawn starting from their upper left corner. Therefore, we subtract half of the square's size when calculating the `x` position. The `y` position starts at a negative value of the square's size, so it starts completely outside the screen.

Update square positions

Now we can iterate through the vector using a for loop and update the Y position using the square's speed and the variable `delta_time`. This will make the squares move downwards across the screen.

```
for square in &mut squares {
    square.y += square.speed * delta_time;
}
```

Remove invisible squares

Next, we need to clean up all the squares that have moved off the bottom of the screen since it's unnecessary to draw things that are not visible. We'll use the `retain()` method on the vector, which takes a function that determines whether elements should be kept. We'll check if the square's `y` value is still less than the height of the window plus the size of the square.

```
squares.retain(|square| square.y < screen_height() + square.size);
```

Draw the squares

Finally, we add a `for` loop that iterates over the `squares` vector and uses the function `draw_rectangle()` to draw a rectangle at the updated position and with the correct size. Since rectangles are drawn with x and y from the top-left corner and our coordinates are based on the center of the square, we use some mathematics to calculate where they should be placed. The size is used twice, once for the width of the square and once for the height. We set the color to `GREEN` so that all squares will have a green color.

```
for square in &squares {
    draw_rectangle(
        square.x - square.size / 2.0,
        square.y - square.size / 2.0,
        square.size,
        square.size,
        GREEN,
    );
}
```

> **Fancier rectangles available**
>
> It's also possible to use the function `draw_rectangle_ex()` (https://docs.rs/macroquad/latest/macroquad/shapes/fn.draw_rectangle_ex.html) that uses the struct `DrawTextureParams` (https://docs.rs/macroquad/latest/macroquad/shapes/struct.DrawRectangleParams.html) instead of a color. In addition to setting color, it can be used to set `rotation` and `offset` of the rectangle.

Challenge: Multiple colors

Try setting a different color for each square by using the method `choose()` on vectors from Macroquad's ChooseRandom trait (https://docs.rs/macroquad/latest/macroquad/rand/trait.ChooseRandom.html), which returns a random element from the vector.

Collisions

To make the game more exciting, let's add some conflict. If our hero, the brave yellow circle, collides with a square, the game will be over and has to be restarted.

After we have drawn the circle and all squares, we'll add a check to see if any square touches the circle. If it does, we'll display the text "`GAME OVER!`" in capital letters and wait for the player to press the space key. When the player presses space, we'll reset the vector with squares and move the circle back to the center of the screen.

Implementation

Collision function

We expand the `Shape` struct with an implementation that contains the method `collides_with()` to check if it collides with another `Shape`. This method uses the `overlaps()` helper method from Macroquad's `Rect` (https://docs.rs/macroquad/latest/macroquad/math/struct.Rect.html) struct. We also create a helper method called `rect()` that creates a `Rect` from our Shape.

> **The Macroquad Rect struct**
>
> There are many methods on `Rect` to do calculations on rectangles, such as `contains()`, `intersect()`, `scale()`, `combine_with()` and `move_to()`.

```
impl Shape {
    fn collides_with(&self, other: &Self) -> bool {
        self.rect().overlaps(&other.rect())
    }

    fn rect(&self) -> Rect {
        Rect {
            x: self.x - self.size / 2.0,
            y: self.y - self.size / 2.0,
            w: self.size,
            h: self.size,
        }
    }
}
```

> **Rect origin**
>
> The origin of Macroquad's `Rect` is also from the top left corner, so we must subtract half its size from both `x` and `y`.

Is it game over?

Let's add a boolean variable called `gameover` to the start of the main loop to keep track of whether the player has died.

```
    let mut gameover = false;
```

Since we don't want the circle and squares to move while it's game over, the movement code is wrapped in an `if` statement that checks if the `gameover` variable is `false`.

```
if !gameover {
    ...
}
```

Collision

After the movement code, we add a check if any square collides with the circle. We use the method `any()` on the iterator for the vector `squares` and check if any square collides with our hero circle. If a collision occurs, we set the variable `gameover` to true.

```
if squares.iter().any(|square| circle.collides_with(square)) {
    gameover = true;
}
```

Challenge: Circle collision

The collision code assumes that the circle is a square. Try writing code that takes into account that the circle does not entirely fill the square.

Reset the game

If the `gameover` variable is `true` and the player has just pressed the space key, we clear the vector `squares` using the `clear()` method and reset the `x` and `y` coordinates of `circle` to the center of the screen. Then, we set the variable `gameover` to `false` so that the game can start over.

```
if gameover && is_key_pressed(KeyCode::Space) {
    squares.clear();
    circle.x = screen_width() / 2.0;
    circle.y = screen_height() / 2.0;
    gameover = false;
}
```

Key press difference

The difference between the functions `is_key_down()` and `is_key_pressed()` is that the latter only checks if the key was pressed during the current frame, while the former returns true for all frames from when the button was pressed and then held down. An experiment you can do is to use `is_key_pressed()` to control the circle.

There's also a function called `is_key_released()` which checks if the key was released during the current frame.

Display GAME OVER

Finally, we draw the text "GAME OVER!" in the middle of the screen after the circle and squares have been drawn, but only if the variable `gameover` is `true`. Macroquad does not have any feature to decide which things will be drawn on top of other things. Each thing drawn will be drawn on top of all other things drawn earlier during the the same frame.

Extended text parameters

It's also possible to use the function `draw_text_ex()` (https://docs.rs/macroquad/latest/macroquad/text/fn.draw_text_ex.html) which takes a `DrawTextParams` struct (https://docs.rs/macroquad/latest/macroquad/text/struct.TextParams.html) instead of `font_size` and `color`. Using that struct it's possible to set more parameters such as `font`, `font_scale`, `font_scale_aspect` and `rotation`.

```
if gameover {
    let text = "GAME OVER!";
    let text_dimensions = measure_text(text, None, 50, 1.0);
    draw_text(
        text,
        screen_width() / 2.0 - text_dimensions.width / 2.0,
        screen_height() / 2.0,
        50.0,
        RED,
    );
}
```

Challenge: Calculate midpoint

Since `draw_text()` is based on the text's baseline, the text won't appear exactly in the center of the screen. Try using the `offset_y` and `height` fields from `text_dimensions` to calculate the text's midpoint. Macroquad's example text measures (https://github.com/not-fl3/macroquad/blob/master/examples/text_measures.rs) can provide tips on how it works.

Bullet hell

It is slightly unfair that our poor circle isn't able to defend itself against the terrifying squares. So it's time to implement the ability for the circle to shoot bullets.

Implementation

Dead or alive?

To keep track of which squares have been hit by bullets we add the field `collided` of the type `bool` to the struct `Shape`.

```
struct Shape {
    size: f32,
    speed: f32,
    x: f32,
    y: f32,
    collided: bool,
}
```

Keeping track

We need another vector to keep track of all the bullets. For simplicity's sake we'll call it `bullets`. Add it after the `squares` vector. Here we'll also set the type of the elements to ensure that the Rust compiler knows what type it is before we have added anything to it. We'll use the struct `Shape` for the bullets as well.

```
    let mut bullets: Vec<Shape> = vec![];
```

Shoot bullets

After the circle has moved we'll add a check if the player has pressed the space key and add a bullet to the `bullets` vector. The `x` and `y` coordinates of the bullet are set to the same values as for the circle, and the speed is set to twice that of the circle.

```
        if is_key_pressed(KeyCode::Space) {
            bullets.push(Shape {
                x: circle.x,
                y: circle.y,
                speed: circle.speed * 2.0,
                size: 5.0,
                collided: false,
            });
        }
```

> **Only one shot**
>
> Note that we're using the function `is_key_pressed()` which only returns true during the frame when the key was first pressed.

Since we added a new field to the `Shape` struct we'll need to set it when we create a square.

```
    squares.push(Shape {
        size,
        speed: rand::gen_range(50.0, 150.0),
        x: rand::gen_range(size / 2.0, screen_width() - size / 2.0),
        y: -size,
        collided: false,
    });
```

Move bullets

We don't want the bullets to be stationary mines, so we'll have to loop over the `bullets` vector and move them in the `y` direction. Add the following code after the code that moves the squares.

```
for square in &mut squares {
    square.y += square.speed * delta_time;
}
for bullet in &mut bullets {
    bullet.y -= bullet.speed * delta_time;
}
```

Remove bullets and squares

Make sure to remove the bullets that have exited the screen in the same way that the squares are removed.

```
bullets.retain(|bullet| bullet.y > 0.0 - bullet.size / 2.0);
```

Now it is time to remove all the squares and bullets that have collided. It can be done with the `retain` method on the vectors which takes a predicate that should return `true` if the element should be kept. We'll just check whether the `collided` field on the struct is false. Do the same thing for both the `squares` and the `bullets` vectors.

```
squares.retain(|square| !square.collided);
bullets.retain(|bullet| !bullet.collided);
```

Collision

After the check if the circle has collided with a square we'll add another check if any of the squares have been hit by a bullet. We'll set the field `collided` to true for both the square and the bullet so that they can be removed.

```
for square in squares.iter_mut() {
    for bullet in bullets.iter_mut() {
        if bullet.collides_with(square) {
            bullet.collided = true;
            square.collided = true;
        }
    }
}
```

Clear bullets

When the game is over we also have to clear the `bullets` vector so that all the bullets are removed when a new game is started.

```
if gameover && is_key_pressed(KeyCode::Space) {
    squares.clear();
    bullets.clear();
    circle.x = screen_width() / 2.0;
    circle.y = screen_height() / 2.0;
    gameover = false;
}
```

Draw bullets

Before the circle is drawn we'll draw all the bullets that the player has shot. This ensures that they are drawn behind all the other shapes.

```
for bullet in &bullets {
    draw_circle(bullet.x, bullet.y, bullet.size / 2.0, RED);
}
```

> **Outlined circle**
>
> The is another function called `draw_circle_lines()` (https://docs.rs/macroquad/latest/macroquad/shapes/fn.draw_circle_lines.html) that can be used to draw a circle with just the outline.

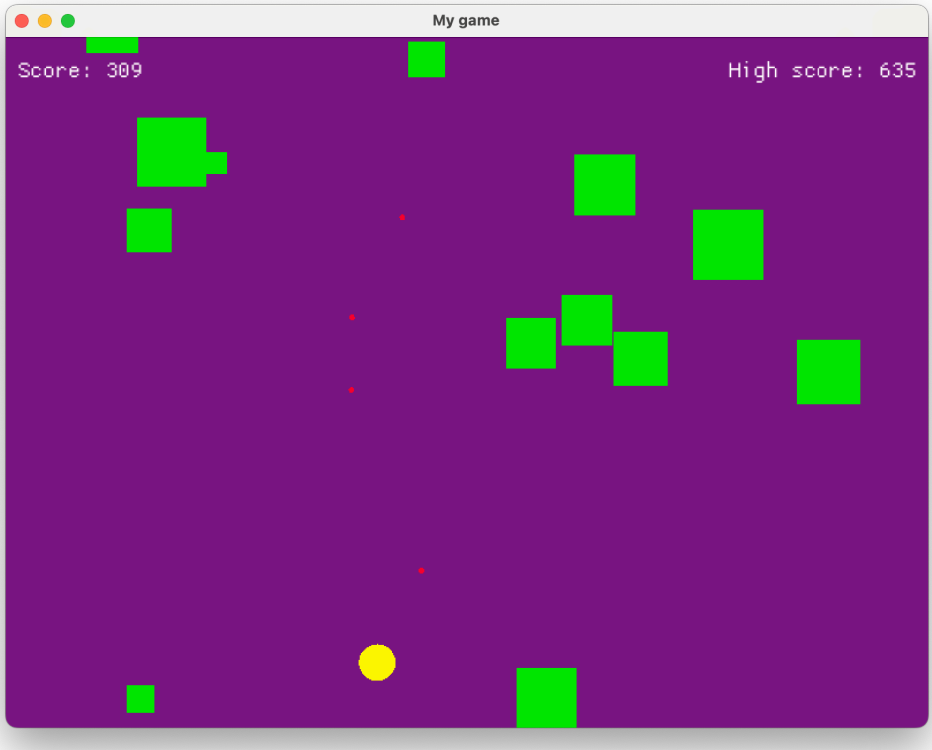
This is all the code that is needed for the circle to be able to shoot down all the fearsome squares.

Challenge: Bullet reloading

To increase the difficulty it's possible to add a minimum time for reloading between each shot. Try using the function `get_time()` (https://docs.rs/macroquad/latest/macroquad/time/fn.get_time.html) to save when the last shot was fired and compare it with the current time. Only add a bullet if the difference is above a certain threshold.

Another possibility is to only allow a specific number of bullets on the screen at the same time.

Points



What is a game without points and a high score? Now that the circle can shoot down the squares it is time to add some points. Every square that is shot down will add to the score, where bigger squares will be worth more points. The current score will be shown on the screen, as well as the highest score achieved.

> **Is bigger better?**
>
> Bigger squares could be worth more because they contain more resources. It is also possible to make bigger circles harder to kill by requiring multiple hits to destroy.

If the current score is the highest score when the game is over, it will be written to a file on disk so that it can be read each time the game is started. This will only work if the game is played on desktop as the WebAssembly version doesn't have access to the file system. It would be possible to store the

high score in the browser storage, but that won't be covered here to keep the implementation simple.

Implementation

Import module

To be able to read and write files we need to import the std::fs module (https://doc.rust-lang.org/std/fs/index.html) from the Rust standard library. Add this line directly below the line to import Macroquad at the top of the file.

```
use std::fs;
```

New variables

We will need two new variables, `score` and `high_score`, to keep track of the player's points as well as the highest score ever achieved. We'll use the function `fs::read_to_string()` to read the file `highscore.dat` from disk. The points stored in the file need to be converted to `u32` with `i.parse::<u32>()`. If anything goes wrong, if the file doesn't exist or it contains something other than a number, the number `0` will be returned instead.

```
    let mut score: u32 = 0;
    let mut high_score: u32 = fs::read_to_string("highscore.dat")
        .map_or(Ok(0), |i| i.parse::<u32>())
        .unwrap_or(0);
```

> **Writing to disk**
>
> We're writing the points directly to the computers hard drive, which will not work if the game has been compiled to WebAssembly and is run on a web page. This will be treated as if the file doesn't exist.
>
> It could be possible to use the browser's storage, or sending the score to a web server, but that is not covered by this guide.

Updating the high score

If the circle collides with a square we'll check if the current score is higher than the high score. If it is higher, we'll update the high score and store the new high score to the file `highscore.dat`.

```
if squares.iter().any(|square| circle.collides_with(square)) {
    if score == high_score {
        fs::write("highscore.dat", high_score.to_string()).ok();
    }
    gameover = true;
}
```

> **Reading files on web**
>
> Macroquad supports reading files when the game is run on a web page. We could use the function `load_string()` (https://docs.rs/macroquad/latest/macroquad/file/fn.load_string.html) to load the high score instead. But since it isn't possible to save the file, this isn't particularly useful in this case.

Increasing the score

When a bullet hits a square, we'll increase the current score based on the size of the square. After that we'll update the `high_score` if the current `score` is higher.

```
if bullet.collides_with(square) {
    bullet.collided = true;
    square.collided = true;
    score += square.size.round() as u32;
    high_score = high_score.max(score);
}
```

Resetting the score

When a new game is started, we need to set the `score` variable to `0`.

```
if gameover && is_key_pressed(KeyCode::Space) {
    squares.clear();
    bullets.clear();
    circle.x = screen_width() / 2.0;
    circle.y = screen_height() / 2.0;
    score = 0;
    gameover = false;
}
```

Displaying scores

Finally, we'll display the `score` and `high_score` on the screen. We'll display the `score` in the top left corner of the screen. To be able to display the high score in the top right corner we'll use the

function `measure_text()` (https://docs.rs/macroquad/latest/macroquad/text/fn.measure_text.html) to calculate how far from the right edge of the screen the text should be displayed.

To ensure that the dimensions are correct we must use the same arguments for both `measure_text()` and `draw_text()`. The arguments for these functions are `text`, `font`, `font_size` and `font_scale`. Since we aren't setting any specific font or scaling the size of the text, we'll use `None` as the value for `font`, and `1.0` as `font_scale`. The `font_size` can be set to `25.0`.

```
draw_text(
    format!("Score: {}", score).as_str(),
    10.0,
    35.0,
    25.0,
    WHITE,
);
let highscore_text = format!("High score: {}", high_score);
let text_dimensions = measure_text(highscore_text.as_str(), None, 25, 1.0);
draw_text(
    highscore_text.as_str(),
    screen_width() - text_dimensions.width - 10.0,
    35.0,
    25.0,
    WHITE,
);
```

Measure text

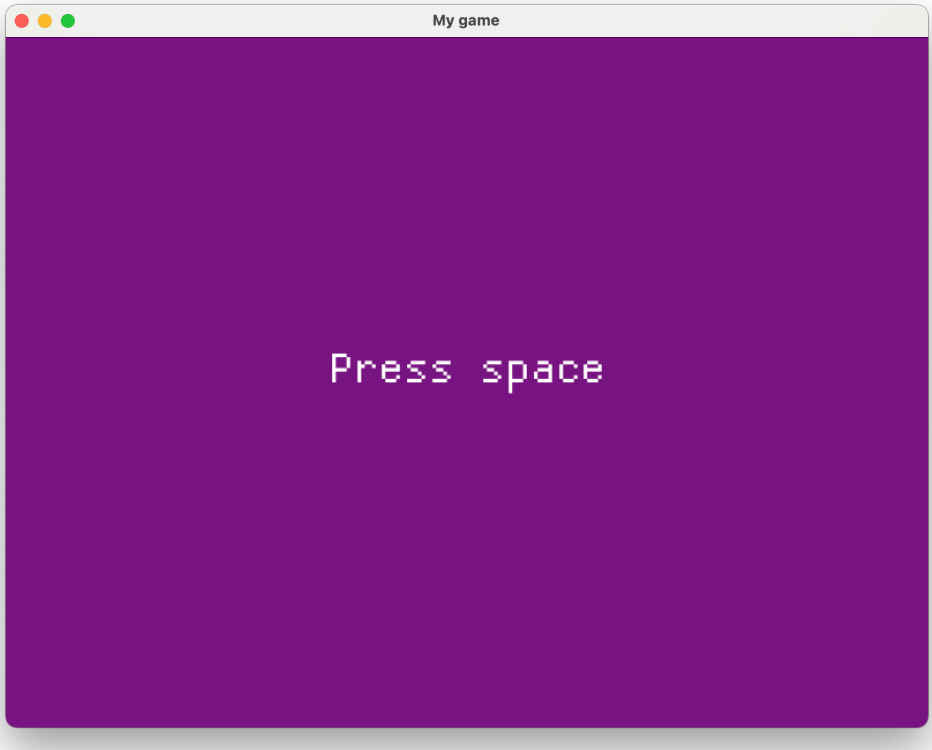
The function `measure_text()` returns the struct `TextDimensions` (https://docs.rs/macroquad/latest/macroquad/text/struct.TextDimensions.html) which contains the fields `width`, `height`, and `offset_y`.

Run the game and try to get a high score!

Challenge: High score message

Try writing a congratulations message below the "GAME OVER" text if the player reached a high score.

Game state



Before we add any more functionality to our game it's time for some refactoring. To make it easier to keep track of the game state we'll add an enum called `GameState` with variants to differentiate between the game being played and the game being over. This will allows us to remove the `gameover` variable, and we can add states for showing a start menu and pausing the game.

Implementation

Game state enum

Begin by adding an enum called `GameState` below the `Shape` implementation. It should contain all four possible game states: `MainMenu`, `Playing`, `Paused`, and `GameOver`.

```
enum GameState {
    MainMenu,
    Playing,
    Paused,
    GameOver,
}
```

Game state variable

Replace the line that declares the `gameover` variable with a line that instantiates a `game_state` variable set to `GameState::MainMenu`.

```
    let mut game_state = GameState::MainMenu;
```

Match on GameState

We'll replace the old code in the game loop with code that uses the `match` control flow construct on the `game_state` variable. It has to match on all four states in the enum. Later on we'll add back code from the earlier chapter within the matching arms. Keep the call to clearing the screen at the start of the loop, and the call to `next_frame().await` at the end.

```
        clear_background(DARKPURPLE);

        match game_state {
            GameState::MainMenu => {
                ...
            }
            GameState::Playing => {
                ...
            }
            GameState::Paused => {
                ...
            }
            GameState::GameOver => {
                ...
            }
        }

        next_frame().await
```

Main menu

Now let's add back code into the match arms to handle each game state. When the game is started, the state will be `GameState::MainMenu`. We'll start by quitting the game if the `Escape` key is pressed. If the player presses the space key we'll set the `game_state` to the new state `GameState::Playing`.

We'll also reset all the game variables. We will also draw the text "Press space" in the middle of the screen.

```
GameState::MainMenu => {
    if is_key_pressed(KeyCode::Escape) {
        std::process::exit(0);
    }
    if is_key_pressed(KeyCode::Space) {
        squares.clear();
        bullets.clear();
        circle.x = screen_width() / 2.0;
        circle.y = screen_height() / 2.0;
        score = 0;
        game_state = GameState::Playing;
    }
    let text = "Press space";
    let text_dimensions = measure_text(text, None, 50, 1.0);
    draw_text(
        text,
        screen_width() / 2.0 - text_dimensions.width / 2.0,
        screen_height() / 2.0,
        50.0,
        WHITE,
    );
},
```

Playing the game

Let's add back the code for playing the game to the matching arm for the state `GameState::Playing`. It's the same code as most of the game loop from the last chapter. However, don't add back the code that handles Game Over as it will be added in the matching arm for the `GameState::GameOver`.

We'll also add a code that checks if the player presses the `Escape` key and change the state to `GameState::Paused`. This will ensure that the game will be paused in the next iteration of the game loop.

```
GameState::Playing => {
    let delta_time = get_frame_time();
    if is_key_down(KeyCode::Right) {
        circle.x += MOVEMENT_SPEED * delta_time;
    }
    if is_key_down(KeyCode::Left) {
        circle.x -= MOVEMENT_SPEED * delta_time;
    }
    if is_key_down(KeyCode::Down) {
        circle.y += MOVEMENT_SPEED * delta_time;
    }
    if is_key_down(KeyCode::Up) {
        circle.y -= MOVEMENT_SPEED * delta_time;
    }
    if is_key_pressed(KeyCode::Space) {
        bullets.push(Shape {
            x: circle.x,
            y: circle.y,
            speed: circle.speed * 2.0,
            size: 5.0,
            collided: false,
        });
    }
    if is_key_pressed(KeyCode::Escape) {
        game_state = GameState::Paused;
    }

    // Clamp X and Y to be within the screen
    circle.x = clamp(circle.x, 0.0, screen_width());
    circle.y = clamp(circle.y, 0.0, screen_height());

    // Generate a new square
    if rand::gen_range(0, 99) >= 95 {
        let size = rand::gen_range(16.0, 64.0);
        squares.push(Shape {
            size,
            speed: rand::gen_range(50.0, 150.0),
            x: rand::gen_range(size / 2.0, screen_width() - size / 2.0),
            y: -size,
            collided: false,
        });
    }

    // Movement
    for square in &mut squares {
        square.y += square.speed * delta_time;
    }
    for bullet in &mut bullets {
        bullet.y -= bullet.speed * delta_time;
    }

    // Remove shapes outside of screen
    squares.retain(|square| square.y < screen_height() + square.size);
    bullets.retain(|bullet| bullet.y > 0.0 - bullet.size / 2.0);

    // Remove collided shapes
    squares.retain(|square| !square.collided);
    bullets.retain(|bullet| !bullet.collided);
```

```
                // Check for collisions
                if squares.iter().any(|square| circle.collides_with(square)) {
                    if score == high_score {
                        fs::write("highscore.dat", high_score.to_string()).ok();
                    }
                    game_state = GameState::GameOver;
                }
                for square in squares.iter_mut() {
                    for bullet in bullets.iter_mut() {
                        if bullet.collides_with(square) {
                            bullet.collided = true;
                            square.collided = true;
                            score += square.size.round() as u32;
                            high_score = high_score.max(score);
                        }
                    }
                }

                // Draw everything
                for bullet in &bullets {
                    draw_circle(bullet.x, bullet.y, bullet.size / 2.0, RED);
                }
                draw_circle(circle.x, circle.y, circle.size / 2.0, YELLOW);
                for square in &squares {
                    draw_rectangle(
                        square.x - square.size / 2.0,
                        square.y - square.size / 2.0,
                        square.size,
                        square.size,
                        GREEN,
                    );
                }
                draw_text(
                    format!("Score: {}", score).as_str(),
                    10.0,
                    35.0,
                    25.0,
                    WHITE,
                );
                let highscore_text = format!("High score: {}", high_score);
                let text_dimensions = measure_text(highscore_text.as_str(), None, 25,
1.0);
                draw_text(
                    highscore_text.as_str(),
                    screen_width() - text_dimensions.width - 10.0,
                    35.0,
                    25.0,
                    WHITE,
                );
            },
```

Pause the game

Many games have the option to pause the action, so we'll add support for that in our game, too. When the game is paused, we'll check if the player presses the `Space` key and change the game

state to `GameState::Playing` so that the game can continue. We'll also draw a text on the screen showing that the game is paused.

The changed game state will only come into effect in the next iteration of the game loop, so even if it has been changed we need to display the text during the current frame.

```
GameState::Paused => {
    if is_key_pressed(KeyCode::Space) {
        game_state = GameState::Playing;
    }
    let text = "Paused";
    let text_dimensions = measure_text(text, None, 50, 1.0);
    draw_text(
        text,
        screen_width() / 2.0 - text_dimensions.width / 2.0,
        screen_height() / 2.0,
        50.0,
        WHITE,
    );
},
```

Game Over

Finally we will handle what happens when the game is over. If the player presses the space bar we'll change the state to `GameState::MainMenu` to allow the player to start a new game or quit the game. We'll also draw the "GAME OVER!" text to the screen as we did in the last chapter.

```
GameState::GameOver => {
    if is_key_pressed(KeyCode::Space) {
        game_state = GameState::MainMenu;
    }
    let text = "GAME OVER!";
    let text_dimensions = measure_text(text, None, 50, 1.0);
    draw_text(
        text,
        screen_width() / 2.0 - text_dimensions.width / 2.0,
        screen_height() / 2.0,
        50.0,
        RED,
    );
},
```

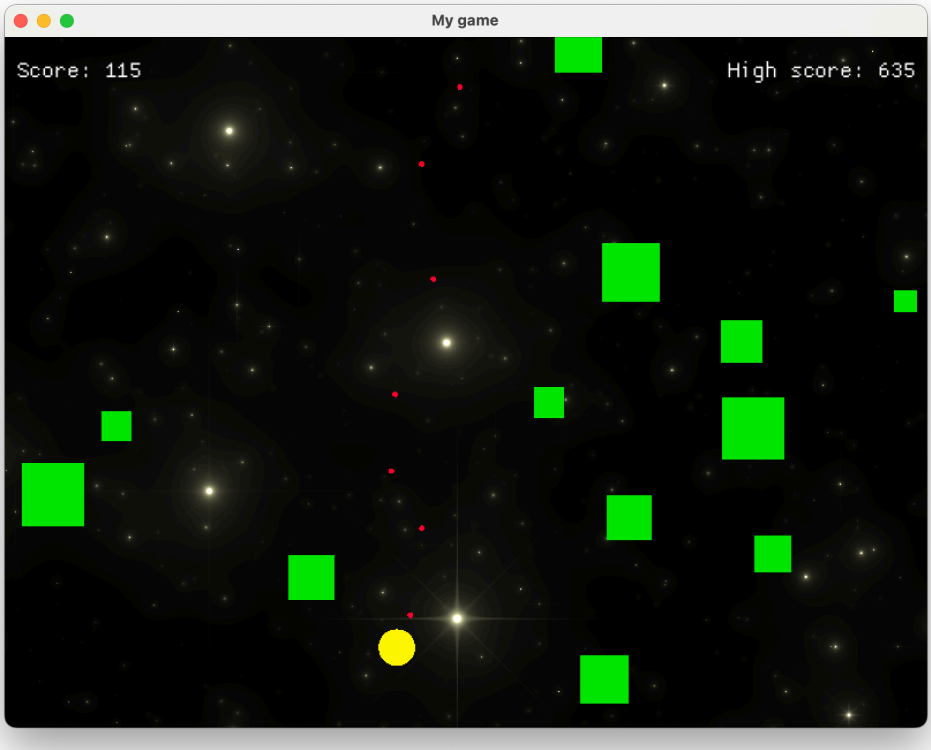
> **Separate game states**
>
> Since the states for `GameState::Playing` and `GameState::GameOver` are separated, the squares and circles will not be shown when the game is paused.

Challenge: Name of the game

Now that we have a main menu, you could come up with a name for your game and display it in a large font at the top of the screen in the state `GameState::MainMenu`.

You could also try drawing all the circles and squares even when the game is paused without moving them.

Starfield shader



The purple background on the screen is starting to feel a bit boring. Instead we'll add something more interesting. We'll use a pixel shader to display a moving starfield in the background. How to implement a shader is outside the scope of this guide, so we'll use one that has already been prepared for us.

In short, a shader is a small program that runs on the GPU of the computer. They are written in a C-like programming language called GLSL. The shader is made up of two parts, a vertex shader and a fragment shader. The vertex shader converts from coordinates in a 3D environment to the 2D coordinates of the screen. Whereas the fragment shader is run for every pixel on the screen to set the variable `gl_FragColor` to define the color that pixel should have. Since our game is entirely in 2D, the vertex shader won't do anything other than setting the position.

Implementation

Shaders

At the top of the `main.rs` file we'll add a vertex shader, the fragment shader will be loaded from a file that we will add later. We'll use the Rust macro `include_str!()` to read the file as a `&str` at compile time. The vertex shader is so short that it can be added directly in the Rust source code.

The most important line in the vertex shader is the line that sets `gl_Position`. For simplicity's sake we'll also set the `iTime` variable that is used by the fragment shader from `_Time.x`. It would also be possible to use `_Time` directly in the fragment shader, but it would mean we have to change it slightly.

```
const FRAGMENT_SHADER: &str = include_str!("starfield-shader.glsl");

const VERTEX_SHADER: &str = "#version 100
attribute vec3 position;
attribute vec2 texcoord;
attribute vec4 color0;
varying float iTime;

uniform mat4 Model;
uniform mat4 Projection;
uniform vec4 _Time;

void main() {
    gl_Position = Projection * Model * vec4(position, 1);
    iTime = _Time.x;
}
";
```

Initialize the shader

In the `main()` function, above the loop, we need to setup a few variables to be able to use the shader. We start by adding the variable `direction_modifier` that will be used to change the direction of the stars horizontally, depending on whether the circle is moved left or right. After that we create a `render_target` to which the shader will be rendered.

Now we can create a `Material` with the vertex shader and the fragment shader using the enum `ShaderSource::Glsl`.

In the parameters we'll also setup two uniforms for the shader that are global variables that we can set for every frame. The uniform `iResolution` will contain the size of the window and `direction_modifier` is used to control the direction of the stars.

```
let mut direction_modifier: f32 = 0.0;
let render_target = render_target(320, 150);
render_target.texture.set_filter(FilterMode::Nearest);
let material = load_material(
    ShaderSource::Glsl {
        vertex: VERTEX_SHADER,
        fragment: FRAGMENT_SHADER,
    },
    MaterialParams {
        uniforms: vec![
            UniformDesc::new("iResolution", UniformType::Float2),
            UniformDesc::new("direction_modifier", UniformType::Float1),
        ],
        ..Default::default()
    },
)
.unwrap();
```

Available uniforms

Macroquad will automatically add some uniforms to all shaders. The available uniforms are `_Time`, `Model`, `Projection`, `Texture`, and `_ScreenTexture`.

Draw the shader

It's now time to change the purple background to our new starfield. Change the line `clear_background(DARKPURPLE);` to the code below.

The first thing we need to do is to set the window resolution to the material uniform `iResolution`. We'll also set the `direction_modifier` uniform to the same value as the corresponding variable.

After this we'll use the function `gl_use_material()` to use the material. Finally we can use the function `draw_texture_ex()` to draw the texture from our `render_target` on the background of the screen. Before we continue we'll restore the shader with the function `gl_use_default_material()` so that it won't be used when drawing the rest of the game.

```
clear_background(BLACK);

material.set_uniform("iResolution", (screen_width(), screen_height()));
material.set_uniform("direction_modifier", direction_modifier);
gl_use_material(&material);
draw_texture_ex(
    &render_target.texture,
    0.,
    0.,
    WHITE,
    DrawTextureParams {
        dest_size: Some(vec2(screen_width(), screen_height())),
        ..Default::default()
    },
);
gl_use_default_material();
```

Controlling the stars

When the player holds down the left or right arrow key we'll add or subtract a value from the variable `direction_modifier` so that the shader can control the movement of the stars. Remember to multiply the value with `delta_time` so that the change is relative to framerate, just like when doing the movement.

```
if is_key_down(KeyCode::Right) {
    circle.x += MOVEMENT_SPEED * delta_time;
    direction_modifier += 0.05 * delta_time;
}
if is_key_down(KeyCode::Left) {
    circle.x -= MOVEMENT_SPEED * delta_time;
    direction_modifier -= 0.05 * delta_time;
}
```

Create the shader file

Now create a file with the name `starfield-shader.glsl` in the `src` directory to contain the fragment shader and add the following code:

```
#version 100

// Starfield Tutorial by Martijn Steinrucken aka BigWings - 2020
// countfrolic@gmail.com
// License Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Unported License.
// From The Art of Code: https://www.youtube.com/watch?v=rvDo9LvfoVE

precision highp float;

varying vec4 color;
varying vec2 uv;
varying float iTime;

uniform vec2 iResolution;
uniform float direction_modifier;

#define NUM_LAYERS 4.

mat2 Rot(float a) {
    float s = sin(a), c = cos(a);
    return mat2(c, -s, s, c);
}

float Star(vec2 uv, float flare) {
    float d = length(uv);
    float m = .05 / d;

    float rays = max(0., 1. - abs(uv.x * uv.y * 1000.));
    m += rays * flare;
    uv *= Rot(3.1415 / 4.);
    rays = max(0., 1. - abs(uv.x * uv.y * 1000.));
    m += rays * .3 * flare;

    m *= smoothstep(1., .2, d);

    return m;
}

float Hash21(vec2 p) {
    p = fract(p * vec2(123.34, 456.21));
    p += dot(p, p + 45.32);
    return fract(p.x * p.y);
}

vec3 StarLayer(vec2 uv) {
    vec3 col = vec3(0);

    vec2 gv = fract(uv) - .5;
    vec2 id = floor(uv);

    float t = iTime * 0.1;
    for (int y = -1; y <= 1; y++) {
        for (int x = -1; x <= 1; x++) {
            vec2 offs = vec2(x, y);

            float n = Hash21(id + offs); // random between 0 and 1
            float size = fract(n * 345.32);
            float star = Star(gv - offs - vec2(n, fract(n * 42.)) + .5, smoothstep(.9,
```

```
1., size) * .6);
            vec3 color = sin(vec3(.8, .8, .8) * fract(n * 2345.2) * 123.2) * .5 + .5;
            color = color * vec3(0.25, 0.25, 0.20);
            star *= sin(iTime * 3. + n * 6.2831) * .5 + 1.;
            col += star * size * color;
        }
    }
    return col;
}

void main()
{
    vec2 uv = (gl_FragCoord.xy - .5 * iResolution.xy) / iResolution.y;
    float t = iTime * .02;

    float speed = 3.0;
    vec2 direction = vec2(-0.25 + direction_modifier, -1.0) * speed;

    uv += direction;
    vec3 col = vec3(0);

    for (float i = 0.; i < 1.; i += 1. / NUM_LAYERS) {
        float depth = fract(i+t);
        float scale = mix(20., .5, depth);
        float fade = depth * smoothstep(1., .9, depth);
        col += StarLayer(uv * scale + i * 453.2) * fade;
    }

    gl_FragColor = vec4(col, 1.0);
}
```

Shader Coding tutorial video

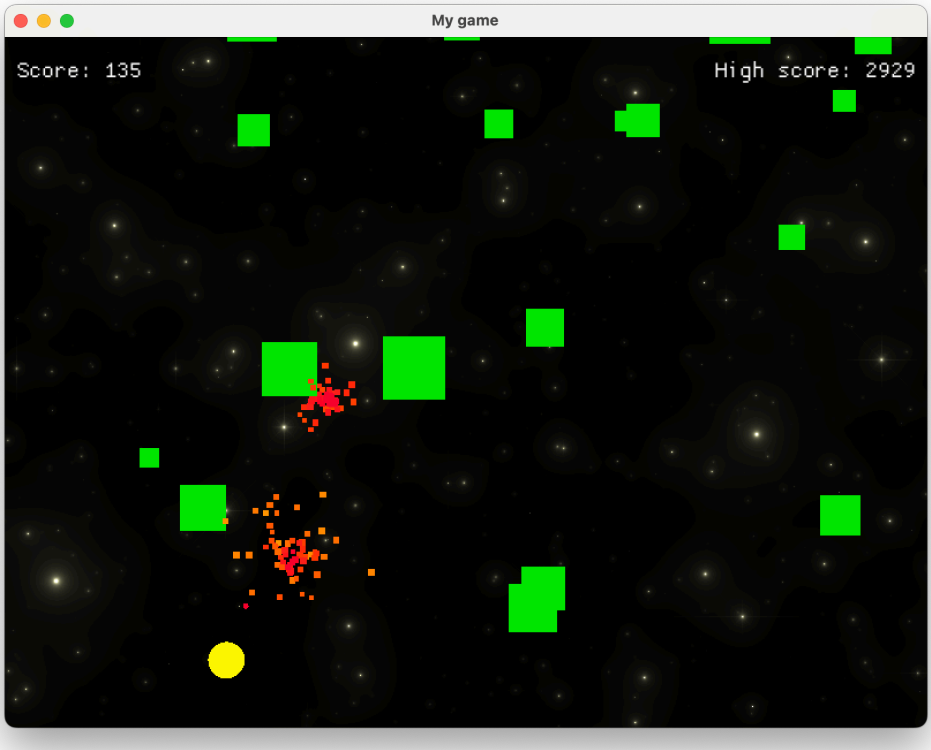
If you want to know how the shader works you can watch the video Shader Coding: Making a starfield (https://youtu.be/rvDo9LvfoVE) by The Art of Code.

Our starfield is now done and the game is starting to look like it takes place in outer space.

Challenge: Star colors

Look at the video Shader Coding: Making a starfield and see if you can change the color and size of the stars.

Particle explosions



We don't want the squares to just disappear when they are hit by a bullet. So now we'll make use of the Macroquad particle system to generate explosions. With the particle system you can easily create and draw many small particles on the screen based on a base configuration. In our case the particles will start from the center of the square and move outwards in all directions. In a later chapter we will add a graphical image to the particles to make it look even more like a real explosion.

Implementation

Add the particle crate

The code for Macroquads particle system is in a separate crate. Start by adding it to the `Cargo.toml` file, either by changing the file by hand, or by running the following command:

```
cargo add macroquad-particles
```

The following line will be added to the `Cargo.toml` file under the heading `[dependencies]`.

```
[package]
name = "my-game"
version = "0.1.0"
edition = "2024"

# See more keys and their definitions at https://doc.rust-
lang.org/cargo/reference/manifest.html

[dependencies]
macroquad = { version = "0.4" }
macroquad-particles = "0.2.2"
```

Bug in macroquad-particles

Version 0.2.2 of macroquad-particles doesn't support the latest version of Macroquad. If you get an error when compiling you can try using both `macroquad` and `macroquad-particles` crates directly from git (https://doc.rust-lang.org/cargo/reference/specifying-dependencies.html#specifying-dependencies-from-git-repositories) by adding the following section to the `Cargo.toml` file.

```
[patch.crates-io]
macroquad = { git = "https://github.com/not-fl3/macroquad" }
macroquad-particles = { git = "https://github.com/not-fl3/macroquad" }
```

Import crate

At the top of `main.rs` we need to import the things we use from the `macroquad_particles` module.

```
use macroquad_particles::{self as particles, ColorCurve, Emitter, EmitterConfig};
```

Particle configuration

We'll use the same configuration for all the explosions, and will only change the size based on the sizes of the squares. Create a function that returns an `EmitterConfig` that can be used to create an `Emitter`. The `Emitter` is a point from where particles can be generated.

```
fn particle_explosion() -> particles::EmitterConfig {
    particles::EmitterConfig {
        local_coords: false,
        one_shot: true,
        emitting: true,
        lifetime: 0.6,
        lifetime_randomness: 0.3,
        explosiveness: 0.65,
        initial_direction_spread: 2.0 * std::f32::consts::PI,
        initial_velocity: 300.0,
        initial_velocity_randomness: 0.8,
        size: 3.0,
        size_randomness: 0.3,
        colors_curve: ColorCurve {
            start: RED,
            mid: ORANGE,
            end: RED,
        },
        ..Default::default()
    }
}
```

Emitter API documentation

There are a lot of different things to configure in an `Emitter`. The fields of `EmitterConfig` (https://docs.rs/macroquad-particles/latest/macroquad_particles/struct.EmitterConfig.html) are described in the documentation of the module `macroquad-particles`.

Vector of explosions

We need another vector to keep track of all the explosions. It includes a tuple with an `Emitter` and the coordinate it should be drawn at.

```
    let mut explosions: Vec<(Emitter, Vec2)> = vec![];
```

When we start a new game, we need to clear the vector of explosions.

```
                if is_key_pressed(KeyCode::Space) {
                    squares.clear();
                    bullets.clear();
                    explosions.clear();
                    circle.x = screen_width() / 2.0;
                    circle.y = screen_height() / 2.0;
                    score = 0;
                    game_state = GameState::Playing;
                }
```

Create an explosion

When a square is hit by a bullet, we'll create a new `Emitter` based on the configuration from `particle_explosion()`, with the addition that the number of particles is based on the size of the square. The coordinates where the particles are generated should be the same as the coordinates of the square.

```
for square in squares.iter_mut() {
    for bullet in bullets.iter_mut() {
        if bullet.collides_with(square) {
            bullet.collided = true;
            square.collided = true;
            score += square.size.round() as u32;
            high_score = high_score.max(score);
            explosions.push((
                Emitter::new(EmitterConfig {
                    amount: square.size.round() as u32 * 2,
                    ..particle_explosion()
                }),
                vec2(square.x, square.y),
            ));
        }
    }
}
```

Removing explosions

When the emitter has finished drawing all the particles, we need to remove them from the `explosions` vector so that we stop trying to draw it. Add the following code below the code that removes squares and bullets.

```
explosions.retain(|(explosion, _)| explosion.config.emitting);
```

Drawing explosions

After drawing all the squares, we can loop through the `explosions` vector and draw them. We only need to send in the coordinates where the particles will be generated, then the emitter will randomize and move all the particles by itself.

```
for (explosion, coords) in explosions.iter_mut() {
    explosion.draw(*coords);
}
```

It's time to try the game to see if there are particle explosions when the squares get hit by bullets.

Challenge: Rocket exhaust

Read the documentation for `EmitterConfig` and try what happens if you change different values. Can you add a particle system that shoots particles out the back of the circle so it looks like a rocket exhaust?

Graphics

It's time to add some graphics to our game to make it look more like a real game. We will do it in three steps so that there won't be too many changes at once. To begin with we'll add code to load textures directly in our `main` function and change the draw function in the game loop. In a later chapter we will look at how to extract the texture loading into a separate function.

Before we make any code changes we need to download all necessary resources. Download this package with graphics and sound and extract it to a directory called `assets` in the root directory of your game.

Assets are available at this address: https://mq.agical.se/assets.zip

All the resources are public domain and are primarily from the website OpenGameArt.org (https://opengameart.org/) which offers lots of different resources to develop games.

The file structure for your game should look like this:

```
.
├── Cargo.lock
├── Cargo.toml
├── README.md
├── assets
│   ├── 8bit-spaceshooter.ogg
│   ├── atari_games.ttf
│   ├── button_background.png
│   ├── button_clicked_background.png
│   ├── enemy-big.png
│   ├── enemy-medium.png
│   ├── enemy-small.png
│   ├── explosion.png
│   ├── explosion.wav
│   ├── laser-bolts.png
│   ├── laser.wav
│   ├── ship.png
│   └── window_background.png
└── src
    ├── main.rs
    └── starfield-shader.glsl
```

Update web publishing

If you chose to setup web publishing of your game to GitHub Pages in the first chapter you will need to update the file `.github/workflows/deploy.yml` to make sure the assets are included when publishing.

The `assets` directory needs to be created:

```
          mkdir -p ./deploy/assets
```

The asset files need to be copied into the `assets` directory:

```
          cp -r assets/ ./deploy/
```

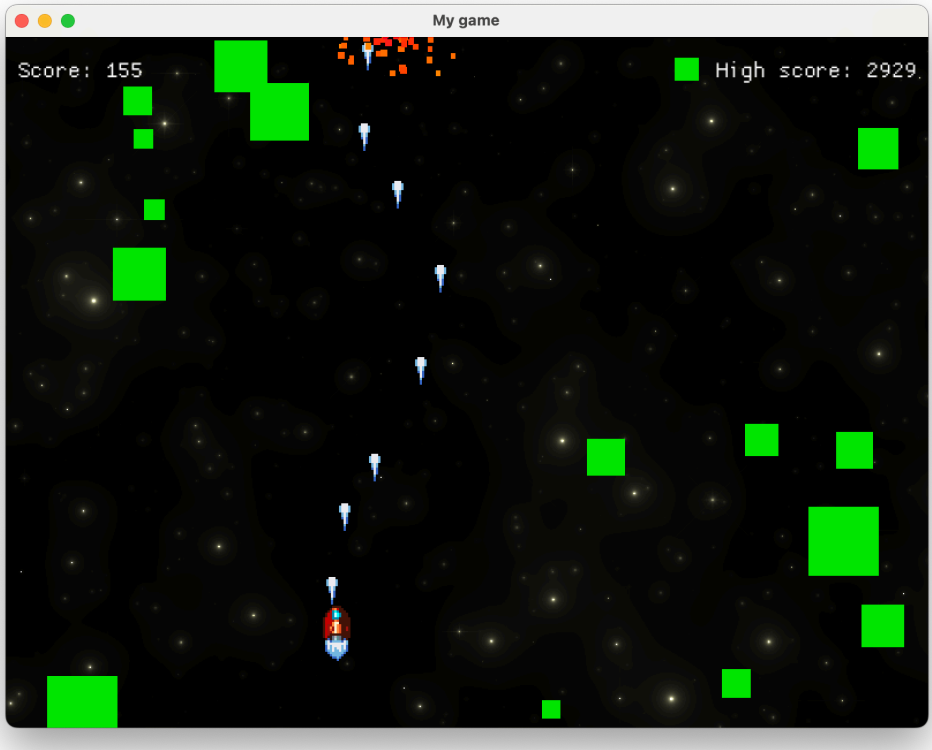
The complete deploy configuration should now look like this:

```yaml
name: Build and Deploy
on:
  push:
    branches:
      - main # If your default branch is named something else, change this

permissions:
  contents: write
  pages: write

jobs:
  build-and-deploy:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout
        uses: actions/checkout@v2

      - name: Install Rust
        uses: actions-rs/toolchain@v1
        with:
          toolchain: stable
          target: wasm32-unknown-unknown
          override: true

      - name: Build
        run: cargo build --release --target wasm32-unknown-unknown

      - name: Prepare Deployment Directory
        run: |
          mkdir -p ./deploy/assets
          cp ./target/wasm32-unknown-unknown/release/my-game.wasm ./deploy/
          cp index.html ./deploy/
          cp -r assets/ ./deploy/

      - name: Deploy
        uses: peaceiris/actions-gh-pages@v3
        with:
          github_token: ${{ secrets.GITHUB_TOKEN }}
          publish_dir: ./deploy
```

Commit your changes and push to GitHub and verify that the game still works on:

- `https://<your-github-account>.github.io/<repository-name>`.

Spaceship and bullets



To begin with we'll add graphics for the spaceship that the player controls. It will be animated with two different sprites and will also have different animations for when the spaceship moves to the left or right. We'll also add a texture with animation for the bullets that the spaceship shoots.

Implementation

Import

The animation support in Macroquad is considered an experimental feature. It might change in a future version of Macroquad. It is not included in the prelude that we have imported, so we will have to import it explicitly.

Import the structs `AnimatedSprite` and `Animation` at the top of `main.rs` file.

```
use macroquad::experimental::animation::{AnimatedSprite, Animation};
```

Configure assets directory

We need to start by defining where Macroquad should read the resources. We'll use the function `set_pc_assets_folder()` that takes the path to the `assets` directory relative to the root directory of the game. This is needed for platforms that might place files in other places and also has the added benefit that we don't need to add the directory name for every file we load.

Add the following code in the `main` function above the game loop:

```
    set_pc_assets_folder("assets");
```

Load textures

Load the image files used for the animation textures of the ship and bullets. Use the function `load_texture()` to load a texture, which takes the name of the file to load. This function is async, because it supports loading files over HTTP in WebAssembly, so we need to call `await` to get the result.

Since loading files can fail, this function will return a `Result`. We will call `expect()` on the result to stop the program if it wasn't possible to load the file. This can happen if the file is missing, or it has wrong read permissions. On WebAssembly it is possible that the HTTP request failed.

After loading the texture we'll set which kind of filter to use when scaling the texture using the method `set_filter()`. We will use the filter `FilterMode::Nearest` because we want to keep the pixelated look of the sprites. This needs to be done on every texture that is loaded. For high resolution textures it would be better to use `FilterMode::Linear` which gives a linear scaling of the texture.

We'll load the file `ship.png` that contains the animations for the spaceship, and the file `laser-bolts.png` that contains animations for two different kinds of bullets.

```
    let ship_texture: Texture2D = load_texture("ship.png").await.expect("Couldn't load
file");
    ship_texture.set_filter(FilterMode::Nearest);
    let bullet_texture: Texture2D = load_texture("laser-bolts.png")
        .await
        .expect("Couldn't load file");
    bullet_texture.set_filter(FilterMode::Nearest);
```

> **Texture2D and Image API**
>
> The images are returned as the struct `Texture2D` (https://docs.rs/macroquad/latest/macroquad/texture/struct.Texture2D.html) that stores the image data in GPU memory. The corresponding struct for images stored in CPU memory is `Image` (https://docs.rs/macroquad/latest/macroquad/texture/struct.Image.html).

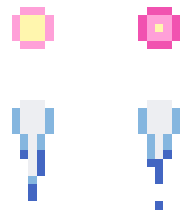
Build a texture atlas

After loading all the textures we'll call the Macroquad function `build_textures_atlas()` that will build an atlas containing all loaded textures. This will ensure that all calls to `draw_texture()` and `draw_texture_ex()` will use the texture from the atlas instead of each separate texture, which is much more efficient. All textures need to be loaded before this function is called.

```
build_textures_atlas();
```

Bullet animation

The image `laser-bolts.png` is composed of four sprites, in two rows. These make up the animations for two different types of bullets. We will name the first one `bullet` and the second one `bolt`. Each animation is one row with two frames each and they should be shown at 12 frames per second. The size of the sprites is 16x16 pixels.



Each animation in a spritesheet is placed in a separate row, with the frames next to each other horizontally. Each `Animation` should have a descriptive name, define which row in the spritesheet it is, how many frames it has, and how many frames should be displayed each second.

Create an `AnimatedSprite` with the `tile_width` and `tile_height` set to `16`, and an array with an `Animation` struct for each of the two rows in the spritesheet. The first one should be named `bullet` and have the row `0` and the second one should have the name `bolt` and the row `1`. Both should have `frames` set to `2` and `fps` set to `12`.

We will only use the second animation, so we'll use the method `set_animation()` to define that we will be using the animation on row `1`.

```
let mut bullet_sprite = AnimatedSprite::new(
    16,
    16,
    &[
        Animation {
            name: "bullet".to_string(),
            row: 0,
            frames: 2,
            fps: 12,
        },
        Animation {
            name: "bolt".to_string(),
            row: 1,
            frames: 2,
            fps: 12,
        },
    ],
    true,
);
bullet_sprite.set_animation(1);
```

Spaceship animation

The spritesheet for the spaceship is in the image `ship.png` and we need to define how the animations in the spritesheet should be displayed. We have to create an `AnimatedSprite` for the ship as well. The size of each frame of the spaceship spritesheet is 16x24 pixels, so we'll set `tile_width` to `16` and `tile_height` to `24`. After that is an array with an `Animation` struct for each animation in the spritesheet that we want to use.

There are five animations available in the spritesheet, with the first one in the top row. We will only use three of the spritesheet animations in our `AnimatedSprite`, the second and fourth row from the top in the spritesheet are unused. The first one is used when flying up or down, so add an `Animation` in the `AnimatedSprite` with `row` defined as `0` as the indexes are 0-based, and the `name` set to `idle`. The ship will keep pointing up regardless of if it moves up or down. The second `Animation` is for moving the spaceship to the left, which will use the row with index `2` in the spritesheet. Finally, the third `Animation` is used when moving the spaceship to the right, and has

the `row` index `4`. There are two `frames` in each `Animation` and the `fps` should be set to 12 frames per second.

Finally we set `playing` to `true` so that the animation will be active.

```
let mut ship_sprite = AnimatedSprite::new(
    16,
    24,
    &[
        Animation {
            name: "idle".to_string(),
            row: 0,
            frames: 2,
            fps: 12,
        },
        Animation {
            name: "left".to_string(),
            row: 2,
            frames: 2,
            fps: 12,
        },
        Animation {
            name: "right".to_string(),
            row: 4,
            frames: 2,
            fps: 12,
        },
    ],
    true,
);
```

Animate direction

For the spaceship we need to set which animation to use based on the direction movement. In the code for moving the spaceship we will add a line where we use the method `set_animation()` on the `ship_sprite`. We start by setting the animation to `0` if it isn't turning in any direction, if it is moving to the right we'll set the animation to `2`, and if it moves to the left we'll set the animation to `1`. These numbers are indexes in the array of `Animation` structs we defined in the `AnimatedSprite` for the spaceship, which means they are 0-based.

```
ship_sprite.set_animation(0);
if is_key_down(KeyCode::Right) {
    circle.x += MOVEMENT_SPEED * delta_time;
    direction_modifier += 0.05 * delta_time;
    ship_sprite.set_animation(2);
}
if is_key_down(KeyCode::Left) {
    circle.x -= MOVEMENT_SPEED * delta_time;
    direction_modifier -= 0.05 * delta_time;
    ship_sprite.set_animation(1);
}
```

Change bullet size

Since the graphics for the bullets are larger than the tiny circle we used to draw for them, we need to change the size and starting position when creating a bullet.

```
if is_key_pressed(KeyCode::Space) {
    bullets.push(Shape {
        x: circle.x,
        y: circle.y - 24.0,
        speed: circle.speed * 2.0,
        size: 32.0,
        collided: false,
    });
}
```

Update animations

In order for Macroquad to animate the textures, we need to call the method `update()` on every sprite inside our game loop. Add the following two lines below the code that updates the positions of enemies and bullets.

```
for square in &mut squares {
    square.y += square.speed * delta_time;
}
for bullet in &mut bullets {
    bullet.y -= bullet.speed * delta_time;
}

ship_sprite.update();
bullet_sprite.update();
```

Draw bullet animations

Now we can use the function `draw_texture_ex` to draw each frame of the animation. Remove the lines that draw a circle for each bullet and insert instead the code below. First we call the method `frame()` on the `bullet_sprite` to get the current animation frame and set it to the variable `bullet_frame`.

Inside the loop that draws all the bullets we'll call `draw_texture_ex` to draw the bullet frame. It takes the `bullet_texture` as argument, and an `x` and `y` position based on the size of the bullet. We also add the struct `DrawTextureParams` with the fields `dest_size` and `source_rect`. The field `dest_size` defines in which size the texture will be drawn, so we will use a `Vec2` with the size of the bullet for both `x` and `y`. Finally we'll use `bullet_frame.source_rect`, which is a reference to where in the texture the current frame is placed.

```
let bullet_frame = bullet_sprite.frame();
for bullet in &bullets {
    draw_texture_ex(
        &bullet_texture,
        bullet.x - bullet.size / 2.0,
        bullet.y - bullet.size / 2.0,
        WHITE,
        DrawTextureParams {
            dest_size: Some(vec2(bullet.size, bullet.size)),
            source: Some(bullet_frame.source_rect),
            ..Default::default()
        },
    );
}
```

Draw parameters

By using `DrawTextureParams` (https://docs.rs/macroquad/0.3.25/macroquad/texture/struct.DrawTextureParams.html) it is possible to change how the texture should be drawn. It is possible to draw the texture rotated or mirrored with the fields `rotation`, `pivot`, `flip_x`, and `flip_y`.

Draw the spaceship frames

Finally it's time to replace the circle with the texture for the spaceship. It works in the same way as for the bullets. First we'll retrieve the current frame from the animation sprite, and then we'll draw it using `draw_texture_ex()`.

Because the spaceship animation isn't the same size in width and height, we'll use `ship_frame.dest_size` to define which size should be drawn. To make it a bit bigger we'll double the size.

```
let ship_frame = ship_sprite.frame();
draw_texture_ex(
    &ship_texture,
    circle.x - ship_frame.dest_size.x,
    circle.y - ship_frame.dest_size.y,
    WHITE,
    DrawTextureParams {
        dest_size: Some(ship_frame.dest_size * 2.0),
        source: Some(ship_frame.source_rect),
        ..Default::default()
    },
);
```

If everything works correctly, there should be animated graphics for both the spaceship and the bullets when running the game.

Improve loading times

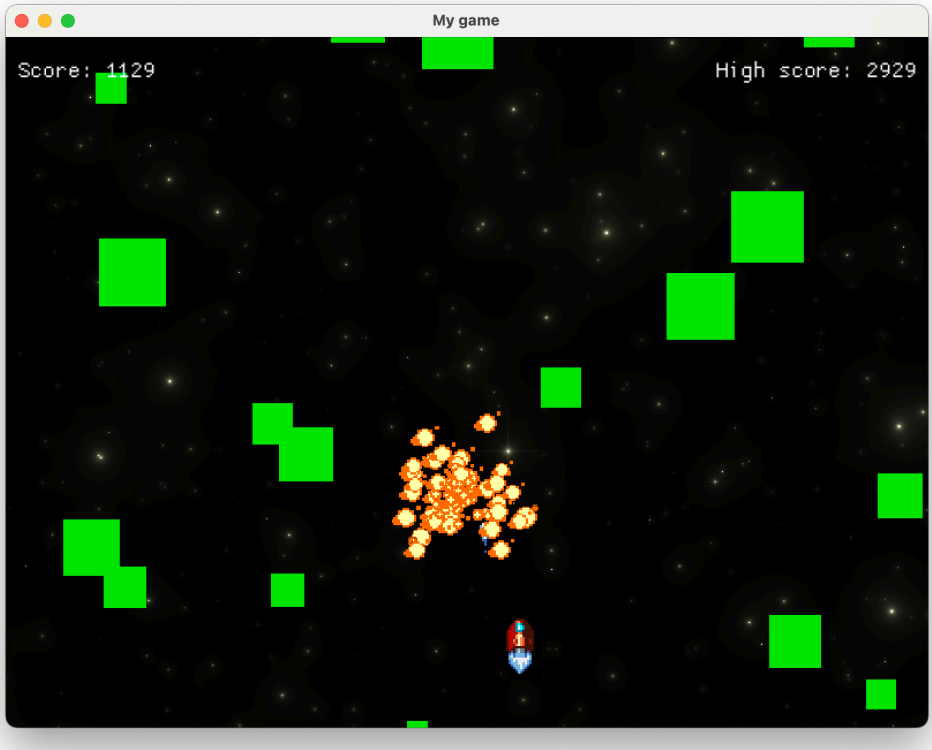
Adding the following snippet at the end of the `Cargo.toml` file will ensure that the assets are loaded much faster when running on a desktop computer.

```
[profile.dev.package.'*']
opt-level = 3
```

Challenge: Turn animation

Try using the two extra spaceship animations to make the ship turn only slightly just when it changes direction and then make it turn fully after a short time.

Graphical explosions



To make the explosions a bit more spectacular we will add graphical textures to the particles.

Implementation

Import

To begin with, we need to update the import of `macroquad_particles` and replace `ColorCurve` with `AtlasConfig`.

```
use macroquad_particles::{self as particles, AtlasConfig, Emitter, EmitterConfig};
```

Update the particle configuration

We need to update the particle configuration for our `particle_explosion` so that it will use `AtlasConfig` to make it use a texture to draw the particles instead of using the `ColorCurve`. We also update the size and lifetime to work better with the graphics.

The `AtlasConfig` describes the layout of the spritesheet when animating particles with a texture. The arguments to `new()` are `n` for columns, `m` for rows, and a `range` for start and end index of the animation. Our spritesheet has five frames in a single row, and we want to use them all for our animation, so we use the values `5`, `1`, and the range `0..`.

```
fn particle_explosion() -> particles::EmitterConfig {
    particles::EmitterConfig {
        local_coords: false,
        one_shot: true,
        emitting: true,
        lifetime: 0.6,
        lifetime_randomness: 0.3,
        explosiveness: 0.65,
        initial_direction_spread: 2.0 * std::f32::consts::PI,
        initial_velocity: 400.0,
        initial_velocity_randomness: 0.8,
        size: 16.0,
        size_randomness: 0.3,
        atlas: Some(AtlasConfig::new(5, 1, 0..)),
        ..Default::default()
    }
}
```

Load textures

Before the line that builds the texture atlas we need to load the texture with the animation for the particle explosion. The file is called `explosion.png`. Don't forget to set the filter on the texture to `FilterMode::Nearest`.

```
    let explosion_texture: Texture2D = load_texture("explosion.png")
        .await
        .expect("Couldn't load file");
    explosion_texture.set_filter(FilterMode::Nearest);
    build_textures_atlas();
```

Add the texture

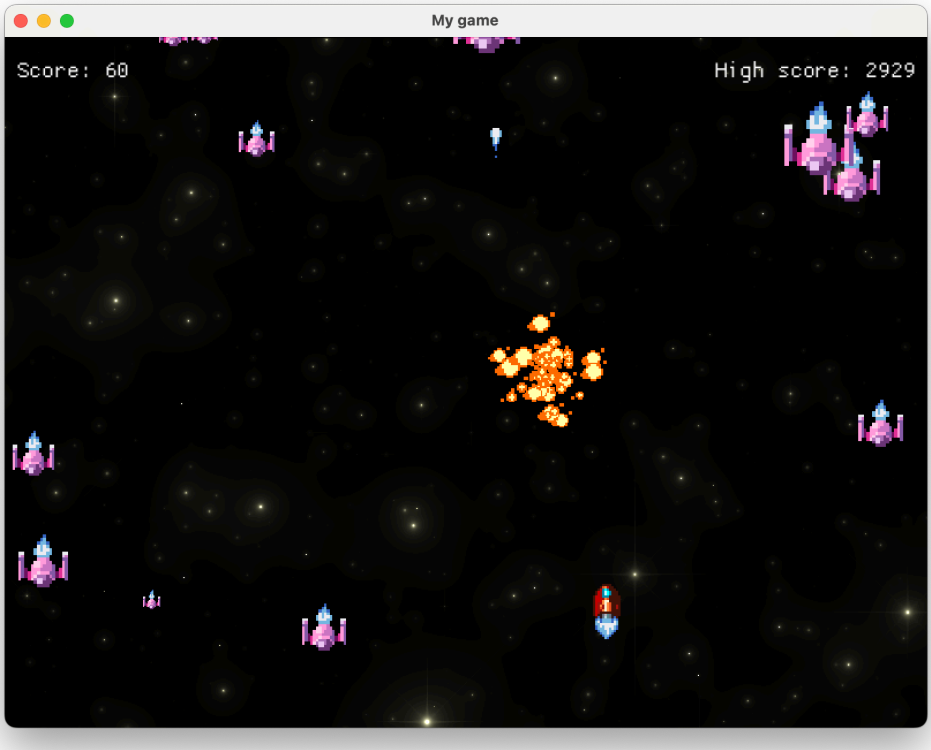
When we create the explosion, we need to add the texture to use. We'll also update the number to get a few more particles. We need to use the method `clone()` on the texture, which is efficient since it is only a pointer to the texture.

```
explosions.push((
    Emitter::new(EmitterConfig {
        amount: square.size.round() as u32 * 4,
        texture: Some(explosion_texture.clone()),
        ..particle_explosion()
    }),
    vec2(square.x, square.y),
));
```

When the game is run, the explosions will be animated with the explosion image instead of colored squares.

> **Challenge: Bigger explosions**
>
>
>
> Change the values of `EmitterConfig` fields based on the size of the enemy that is hit.

Animated enemies



The only thing left is to change the boring squares and replace them with some more exciting graphics. This works the same as when animating the spaceship, we load a texture, create an `AnimatedSprite`, and change how the enemies are drawn to the screen.

Implementation

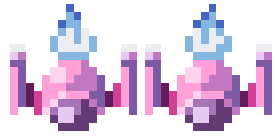
Load the texture

Load the texutre `enemy-small.png` and set the filter mode to `FilterMode::Nearest`.

```
let enemy_small_texture: Texture2D = load_texture("enemy-small.png")
    .await
    .expect("Couldn't load file");
enemy_small_texture.set_filter(FilterMode::Nearest);
build_textures_atlas();
```

Create animation

Create an `AnimatedSprite` to describe the animations in the texture. It is only one animation with two frames. The graphics for the small enemy ships are 16x16 pixels, but the texture has one pixel gutter between the frames to ensure that they don't bleed into each other when we scale the texture.



```
let mut enemy_small_sprite = AnimatedSprite::new(
    17,
    16,
    &[Animation {
        name: "enemy_small".to_string(),
        row: 0,
        frames: 2,
        fps: 12,
    }],
    true,
);
```

Update animation

The enemy sprites need to be updated, add a line with `enemy_small_sprite.update();` after updating the animations for the `ship_sprite` and the `bullet_sprite`.

```
ship_sprite.update();
bullet_sprite.update();
enemy_small_sprite.update();
```

Draw enemy frames

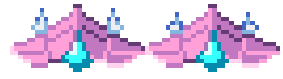
We can now change the drawing of squares to drawing the texture from the current frame of the animation. Vi retrieve the frame from `enemy_small_sprite` and use the `source_rect` in `DrawTextureParams` in the `draw_texture_ex()` call. Since the enemies have a randomized size, we'll use the size of the enemy when setting the `dest_size` and `x` and `y` coordinates.

```
let enemy_frame = enemy_small_sprite.frame();
for square in &squares {
    draw_texture_ex(
        &enemy_small_texture,
        square.x - square.size / 2.0,
        square.y - square.size / 2.0,
        WHITE,
        DrawTextureParams {
            dest_size: Some(vec2(square.size, square.size)),
            source: Some(enemy_frame.source_rect),
            ..Default::default()
        },
    );
}
```

We have now changed to graphics for all the elements of the game, and when you run it now, it should look like a real game.

Challenge: Different enemies

The asset package includes two other enemy spritesheets, `enemy-medium.png` and `enemey-big.png`. Try changing which texture is used for the enemies based on their size.



Music and sound effects

A game doesn't only need graphics to be good, it also needs to sound good. Let's add some music and sound effects to the game.

Implementation

Activate the sound feature

To be able to use sound in Macroquad we need to activate the `audio` feature. This is done by adding `audio` to the list of features in the macroquad dependency in the `Cargo.toml` file.

```
[package]
name = "my-game"
version = "0.1.0"
edition = "2024"

# See more keys and their definitions at https://doc.rust-
lang.org/cargo/reference/manifest.html

[dependencies]
macroquad = { version = "0.4", features = ["audio"] }
macroquad-particles = "0.2.2"
```

Import

The sound module isn't included the Macroquad prelude, so we need to import the `audio` module at the top of the `main.rs` file. The things we need to import are `load_sound`, `play_sound`, `play_sound_once`, and `PlaySoundParams`.

```
use macroquad::audio::{load_sound, play_sound, play_sound_once, PlaySoundParams};
```

Load resources

After all the textures have been loaded, we can load the music and sound effects. There is a file with the music that is called `8bit-spaceshooter.ogg` and two `wav` files with sound effects, `explosion.wav` and `laser.wav`. The music is in the file format Ogg Vorbis which is supported by most, but not all, web browsers.

```
let theme_music = load_sound("8bit-spaceshooter.ogg").await.unwrap();
let sound_explosion = load_sound("explosion.wav").await.unwrap();
let sound_laser = load_sound("laser.wav").await.unwrap();
```

> **Safari sounds**
>
> In order for the music to work on the Safari web browser it has to be converted to `WAV` format. This would make the file very large, so another option is to use a version in `OGG` format and one in `MP3` and select which one to use based on the web browser being used.

Play music

Before the game loop begins we will start playing the music. This is done with the function `play_sound()`, which takes a sound, and the struct `PlaySoundParams` as arguments. In the parameters we set the sound to be played in a loop and with full volume.

```
play_sound(
    &theme_music,
    PlaySoundParams {
        looped: true,
        volume: 1.,
    },
);
```

> **Stop the music**
>
> To stop the music use the function `stop_sound()` which takes the sound as argument.

Play laser sound

When the player is shooting a bullet, we will play the sound effect of a laser blast using the function `play_sound_once()`. This function takes the sound to play as the argument. It is a shortcut instead of using `play_sound()` with a non-looping parameter.

```
bullets.push(Shape {
    x: circle.x,
    y: circle.y - 24.0,
    speed: circle.speed * 2.0,
    size: 32.0,
    collided: false,
});
play_sound_once(&sound_laser);
```

Turn up the volume

It's also possible to set the sound volume per sound using the function `set_sound_volume()` which takes a sound and a number between `0` and `1` as argument.

Play explosion sound

When a bullet hits an enemy, we will play the explosion sound, also using the function `play_sound_once()`.

```
if bullet.collides_with(square) {
    bullet.collided = true;
    square.collided = true;
    score += square.size.round() as u32;
    high_score = high_score.max(score);
    explosions.push((
        Emitter::new(EmitterConfig {
            amount: square.size.round() as u32 * 4,
            texture: Some(explosion_texture.clone()),
            ..particle_explosion()
        }),
        vec2(square.x, square.y),
    ));
    play_sound_once(&sound_explosion);
}
```

You can now start the game, and it should play music and sound effects.

Challenge: Change the volume

It might be a bit intense to start the music at full volume. Try setting the volume lower at the start and increase it once the game starts. Maybe also try to stop the music when the player pauses the game.

Graphical menu

Macroquad has a built-in system to display a graphical user interface where the look can easily be changed using PNG images. We will use this to create a graphical main menu for our game. There will be quite a lot of code to define the look of the UI. However, once that is done, it is very easy to use it.

The menu will have a window centered on the screen with the text "Main menu" in the title bar. Inside the window there will be two buttons, one for "Play" and one for "Quit". The UI will be built using different kinds of widgets such as `label`, `button`, `editbox`, and `combobox`.

Implementation

To begin with we need to import what we need from the `ui` module.

```
use macroquad::ui::{hash, root_ui, Skin};
```

Load resources

After loading the sounds we'll load the font and images used for the UI. There is an image to create the window, `window_background.png`, one image for the buttons, `button_background.png`, and finally an image for when the button is pressed, `button_clicked_background.png`. The images are loaded with the function `load_image()` and binary files with the function `load_file()`. Both images and files are loaded asynchronously and may return errors. This means we will have to call `await` and `unwrap()` to get the files. If we can't load the files needed to display the main menu, we can just exit the program immediately.

```
    let window_background = load_image("window_background.png").await.unwrap();
    let button_background = load_image("button_background.png").await.unwrap();
    let button_clicked_background =
load_image("button_clicked_background.png").await.unwrap();
    let font = load_file("atari_games.ttf").await.unwrap();
```

Create a skin

Before the game loop we need to define how our UI should look. We will build `Style` structs for the window, buttons and texts. After that we will use the styles to create a `Skin`.

We use the function `root_ui()` that will draw widgets last in every frame using a default camera and the coordinate system `(0..screen_width(), 0..screen_height())`.

Window look

To build a style we use a `StyleBuilder` that has helper methods to define all parts of the style. We get access to it by using the method `style_builder()` on `root_ui()`. The values that aren't set will use the same values as the default look.

We will use the method `background()` to set the image used to draw the window. After that we can use `background_margin()` to define which parts of the image that shouldn't change proportion when the window changes size. This is used to ensure that the edges of the window will look good.

The method `margin()` is used to set margins for the content. These values can be negative to draw content to the borders of the window.

```
    let window_style = root_ui()
        .style_builder()
        .background(window_background)
        .background_margin(RectOffset::new(32.0, 76.0, 44.0, 20.0))
        .margin(RectOffset::new(0.0, -40.0, 0.0, 0.0))
        .build();
```

> **StyleBuilder API**
>
> There are many more methods to define styles, these are described in the documentation for Macroquad's `StyleBuilder` (https://docs.rs/macroquad/0.3.25/macroquad/ui/struct.StyleBuilder.html)

Button look

In the definition for buttons we'll use two images. Using `background()` we set the default image for the button, and `background_clicked()` is used to set the image to be displayed while the button is clicked on.

We need to set both `background_margin()` and `margin()` to be able to stretch the image to cover the text inside the button. The look of the text is defined using the methods `font()`, `text_color()`, and `font_size()`.

```
let button_style = root_ui()
    .style_builder()
    .background(button_background)
    .background_clicked(button_clicked_background)
    .background_margin(RectOffset::new(16.0, 16.0, 16.0, 16.0))
    .margin(RectOffset::new(16.0, 0.0, -8.0, -8.0))
    .font(&font)
    .unwrap()
    .text_color(WHITE)
    .font_size(64)
    .build();
```

Text look

Normal text displayed in the interface uses `label_style`. We will use the same font as for the buttons, but in a slightly smaller font size.

```
let label_style = root_ui()
    .style_builder()
    .font(&font)
    .unwrap()
    .text_color(WHITE)
    .font_size(28)
    .build();
```

Define a Skin

We can now create a `Skin` using `window_style`, `button_style`, and `label_style`. We won't define any other styles for the skin as we won't be using them.

We use `push_skin()` to define the current skin that is to be applied. We will only use one skin, but to change between different looks between windows, it's possible to use `push_skin()` and `pop_skin()`.

We will also set the variable `window_size` to define the size of the window.

```
let ui_skin = Skin {
    window_style,
    button_style,
    label_style,
    ..root_ui().default_skin()
};
root_ui().push_skin(&ui_skin);
let window_size = vec2(370.0, 320.0);
```

Macroquad Skin API

It's possible to change the look of more parts of the UI. More information on how to do this can be found in the documentation of the struct Skin (https://docs.rs/macroquad/0.3.25/macroquad/ui/struct.Skin.html).

Build the menu

We can now build a menu by drawing a window with two buttons and a heading. The content of the `GameState::MainMenu` matching arm can be replaced with the code at the end of this chapter.

Start by creating a window using `root_ui().window()`. The function takes an argument that is generated with the macro `hash!`, a position that we'll calculate based on the window size and the screen dimensions, and finally a `Vec2` for the size of the window. Finally it takes a function that is used to draw the content of the window.

Window title

In the window function we start by setting a title for the window with the widget `Label` that we can create using `ui.label()`. The method takes two arguments, a `Vec2` for the position of the label and a string with the text to be displayed. It's possible to set `None` as position, in which case the placement will be relative to the previous widget. We will use a negative `y` position to place the text within the title bar of the window.

Widget builder methods

It's also possible to create widgets by instantiating a struct and using builder methods.

```
widgets::Button::new("Play").position(vec2(45.0, 25.0)).ui(ui);
```

Buttons

After the label we'll add a button to begin playing the game. The method `ui.button()` returns `true` when the button is clicked. We will use this to set the `GameState::Playing` to start a new game.

Then we can create a button with the text "Quit" to exit the game.

```
GameState::MainMenu => {
    root_ui().window(
        hash!(),
        vec2(
            screen_width() / 2.0 - window_size.x / 2.0,
            screen_height() / 2.0 - window_size.y / 2.0,
        ),
        window_size,
        |ui| {
            ui.label(vec2(80.0, -34.0), "Main Menu");
            if ui.button(vec2(65.0, 25.0), "Play") {
                squares.clear();
                bullets.clear();
                explosions.clear();
                circle.x = screen_width() / 2.0;
                circle.y = screen_height() / 2.0;
                score = 0;
                game_state = GameState::Playing;
            }
            if ui.button(vec2(65.0, 125.0), "Quit") {
                std::process::exit(0);
            }
        },
    );
}
```

Macroquad UI API

There are many different widgets that can be used to create interfaces. The list of available widgets can be found in the documentation of the struct `Ui` (https://docs.rs/macroquad/0.3.25/macroquad/ui/struct.Ui.html).

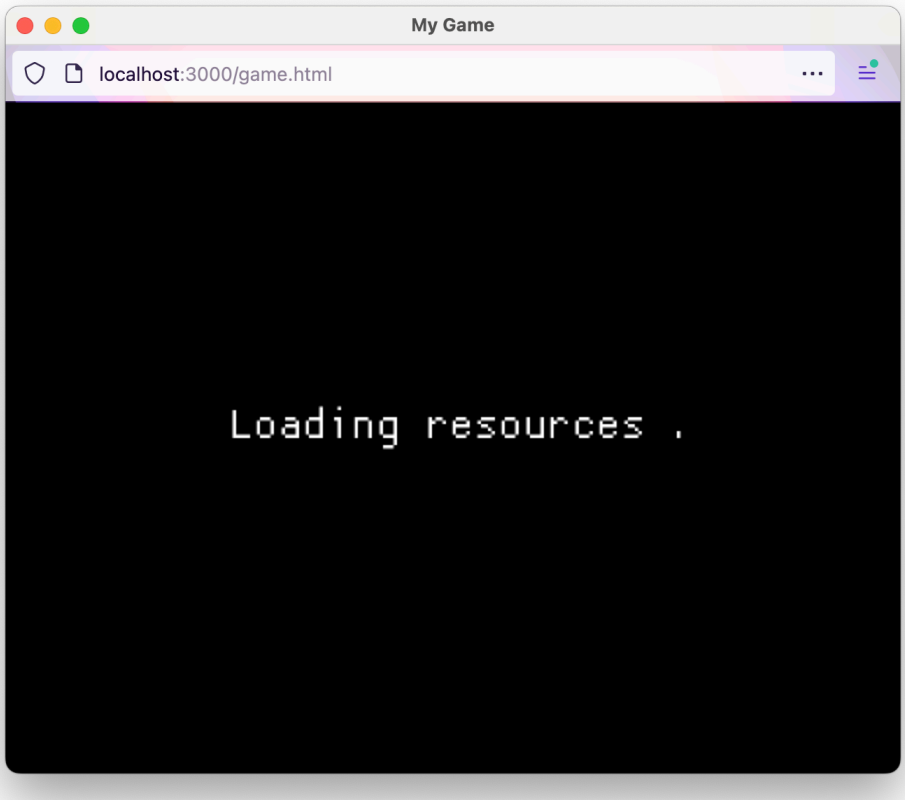
Try the game

When starting the game, a graphical menu will be shown where the player can choose to start a game or quit the program.

Challenge: Switch skins

Try creating a `Skin` of your own from another image and make it possible to switch between the skins while the game is running.

Resources



We're starting to get quite a lot of code in our `main` function so it's time to refactor again to improve the code structure a little.

We'll start by moving all the loading of file assets to a struct. At the same time we will change all the `unwrap()` and `expect()` calls to using the `?` operator to handle error messages.

After that we will make use of a coroutine to load the resources in the background while also displaying a message about loading resources on the screen.

Finally we will use a `Storage` struct to make the resources available in the code without having to send them around to every function where they are needed.

Resources and errors

In this chapter we will refactor our code without adding any new functionality to the game. We do this to build a foundation to be able to add a loading screen during the loading of resources in the web version. We also want to be able to refactor all the drawing to be done by the structs. Finally we will be able to move code away from our `main` function which is starting to get a bit hard to follow.

Implementation

Resources struct

We start by creating a new struct called `Resources` that will contain all the files we load from the file system. Add it above the `main` function. The struct will have a field for every asset loaded.

```
struct Resources {
    ship_texture: Texture2D,
    bullet_texture: Texture2D,
    explosion_texture: Texture2D,
    enemy_small_texture: Texture2D,
    theme_music: Sound,
    sound_explosion: Sound,
    sound_laser: Sound,
    ui_skin: Skin,
}
```

Resources impl

Directly below the `Resources` struct we'll add an implementation block for it. To begin with it will only contain a `new` method that loads all the files and returns an instance of the struct if everything went as expected. We'll reuse the code that used to be in the `main` function to load all the files.

We'll also store the UI `Skin` as a resource so we won't have to return the font and all the images used for it.

The difference in the code is that we've replaced all the `unwrap()` and `expect()` calls to use the `?` operator instead. Using this the error will be returned instead of exiting the program. This means we will be able to handle the error in a single place in our `main` function if we want to. The error message is an enum of the type `macroquad::Error`.

Macroquad errors

The errors available in Macroquad are documented in macroquad::Error (https://docs.rs/macroquad/latest/macroquad/enum.Error.html).

```
impl Resources {
    async fn new() -> Result<Resources, macroquad::Error> {
        let ship_texture: Texture2D = load_texture("ship.png").await?;
        ship_texture.set_filter(FilterMode::Nearest);
        let bullet_texture: Texture2D = load_texture("laser-bolts.png").await?;
        bullet_texture.set_filter(FilterMode::Nearest);
        let explosion_texture: Texture2D = load_texture("explosion.png").await?;
        explosion_texture.set_filter(FilterMode::Nearest);
        let enemy_small_texture: Texture2D = load_texture("enemy-small.png").await?;
        enemy_small_texture.set_filter(FilterMode::Nearest);
        build_textures_atlas();

        let theme_music = load_sound("8bit-spaceshooter.ogg").await?;
        let sound_explosion = load_sound("explosion.wav").await?;
        let sound_laser = load_sound("laser.wav").await?;

        let window_background = load_image("window_background.png").await?;
        let button_background = load_image("button_background.png").await?;
        let button_clicked_background =
load_image("button_clicked_background.png").await?;
        let font = load_file("atari_games.ttf").await?;

        let window_style = root_ui()
            .style_builder()
            .background(window_background)
            .background_margin(RectOffset::new(32.0, 76.0, 44.0, 20.0))
            .margin(RectOffset::new(0.0, -40.0, 0.0, 0.0))
            .build();
        let button_style = root_ui()
            .style_builder()
            .background(button_background)
            .background_clicked(button_clicked_background)
            .background_margin(RectOffset::new(16.0, 16.0, 16.0, 16.0))
            .margin(RectOffset::new(16.0, 0.0, -8.0, -8.0))
            .font(&font)?
            .text_color(WHITE)
            .font_size(64)
            .build();
        let label_style = root_ui()
            .style_builder()
            .font(&font)?
            .text_color(WHITE)
            .font_size(28)
            .build();
        let ui_skin = Skin {
            window_style,
            button_style,
            label_style,
            ..root_ui().default_skin()
        };

        Ok(Resources {
            ship_texture,
            bullet_texture,
            explosion_texture,
            enemy_small_texture,
            theme_music,
            sound_explosion,
```

```
            sound_laser,
            ui_skin,
        })
    }
}
```

Returning errors

To keep things as simple as possible we'll let our `main` function return a result that may be an error. This means we can use the `?` operator in the `main` function as well. If the `main` function returns an error, the game will quit and the error message will be printed on the console.

The standard return value for the `main` function is `()`, which is the Rust unit type that can be used if no value will be returned. Before when the function didn't specify a return value, this was still returned implicitly.

If the last expression in a function ends with a semi colon (`;`) the return value will be skipped and `()` is returned instead.

```
#[macroquad::main("My game")]
async fn main() -> Result<(), macroquad::Error> {
```

> **The Rust unit type**
>
> If you want to know how the Rust unit type works you can find more information in the Rust unit documentation (https://doc.rust-lang.org/std/primitive.unit.html).

Remove `unwrap()`

When loading the material for the shader we used to use the method `unwrap()` which we will now change to the `?` operator to return any error instead. This change is in the last line of the code below.

```
    let material = load_material(
        ShaderSource::Glsl {
            vertex: VERTEX_SHADER,
            fragment: FRAGMENT_SHADER,
        },
        MaterialParams {
            uniforms: vec![
                UniformDesc::new("iResolution", UniformType::Float2),
                UniformDesc::new("direction_modifier", UniformType::Float1),
            ],
            ..Default::default()
        },
    )?;
```

Load resources

We've finally reached the most interesting part of this chapter. It's time to change the code that loads file assets to instead instantiate our `Resources` struct. We add the result to the `resources` variable that we can use later when we need to use a resource.

Note that we use `await` after the `new()` method as it is async. We also use the `?` operator to bubble up any errors.

```
set_pc_assets_folder("assets");
let resources = Resources::new().await?;
```

Update resource usages

Now that we have loaded all the assets with the `Resources` struct we need to update all the places that uses a resource so that they retrieve the asset from it instead. We basically just add `resources.` in front of every resource name.

Game music

```
play_sound(
    &resources.theme_music,
    PlaySoundParams {
        looped: true,
        volume: 1.,
    },
);
```

User interface

Now that we've saved the UI `Skin` in our `Resources` struct we only need to activate it using `root_ui().push_skin()`. We can replace all the lines that builds the UI with a single line.

```
root_ui().push_skin(&resources.ui_skin);
let window_size = vec2(370.0, 320.0);
```

Laser sound

The laser sound needs to use the `resources` variable.

```
    if is_key_pressed(KeyCode::Space) {
        bullets.push(Shape {
            x: circle.x,
            y: circle.y - 24.0,
            speed: circle.speed * 2.0,
            size: 32.0,
            collided: false,
        });
        play_sound_once(&resources.sound_laser);
    }
```

Explosions

We need to update both the texture and the sound for the explosions.

```
                explosions.push((
                    Emitter::new(EmitterConfig {
                        amount: square.size.round() as u32 * 4,
                        texture:
Some(resources.explosion_texture.clone()),
                        ..particle_explosion()
                    }),
                    vec2(square.x, square.y),
                ));
                play_sound_once(&resources.sound_explosion);
```

Bullets

Update the call to drawing bullets to use the texture from `resources`.

```
    for bullet in &bullets {
        draw_texture_ex(
            &resources.bullet_texture,
            bullet.x - bullet.size / 2.0,
            bullet.y - bullet.size / 2.0,
            WHITE,
            DrawTextureParams {
                dest_size: Some(vec2(bullet.size, bullet.size)),
                source: Some(bullet_frame.source_rect),
                ..Default::default()
            },
        );
    }
```

Spaceship

The spaceship also needs to use the texture from `resources`.

```
let ship_frame = ship_sprite.frame();
draw_texture_ex(
    &resources.ship_texture,
    circle.x - ship_frame.dest_size.x,
    circle.y - ship_frame.dest_size.y,
    WHITE,
    DrawTextureParams {
        dest_size: Some(ship_frame.dest_size * 2.0),
        source: Some(ship_frame.source_rect),
        ..Default::default()
    },
);
```

Enemies

When the enemies are drawn, we need to add `resources` as well.

```
for square in &squares {
    draw_texture_ex(
        &resources.enemy_small_texture,
        square.x - square.size / 2.0,
        square.y - square.size / 2.0,
        WHITE,
        DrawTextureParams {
            dest_size: Some(vec2(square.size, square.size)),
            source: Some(enemy_frame.source_rect),
            ..Default::default()
        },
    );
}
```

That's everything that needs to be changed this time. In this chapter we've created a struct that contains all the loaded assets that we use when drawing textures and playing sounds.

> **Challenge: Display errors**
>
>
>
> Instead of just exiting the game when encountering an error you could try to display the error message on the screen using the `draw_text()` function of Macroquad. Remember that the program will then need to keep on running and do nothing but displaying the text.

Try the game

The game should work exactly like before.

> **Cargo dependencies**
>
> Sometimes the cargo dependencies can become out of sync. Some users have experienced this in this chapter. The symptoms are that the buttons in the main menu starts to “glitch” and it requires multiple clicks to press the buttons. A workaround for this issue is to rebuild all the dependencies using `cargo clean`.

Coroutines and Storage

When there are a lot of assets to load, it might take a while to load everything. This is especially true for the WebAssembly version that loads files via HTTP in the browser on a slow internet connection. In these cases we want to display a loading message on the screen instead of just having a completely black screen.

To solve this we will use something called `coroutines`, which will emulate multitasking using the event loop in the browser. For the desktop these will execute immediately instead. This can be used to handle state machines and things that need to be evaluated over time. Using a coroutine we can load all the resources in the background while also drawing to the screen.

Finally we will place the resources in the Macroquad `storage` that is a global persistent storage. It can be used to save game configuration that needs to be available anywhere in the game code without having to send the data around.

> **Experimental features**
>
> Both `coroutines` and `storage` are experimental features of Macroquad and the usage might change in future versions.

Implementation

Importing

Let's start by importing `coroutines::start_coroutine` and `collections::storage` from Macroquad's experimental namespace.

```
use macroquad::experimental::collections::storage;
use macroquad::experimental::coroutines::start_coroutine;
```

Create a new load method

Now we can create a `load()` method in the implementation block for the `Resources` struct. In this method we'll add the code that loads the assets using a coroutine and display a text message on the screen showing that resources are being loaded.

The function `start_coroutine` takes an `async` block and returns a `Coroutine`. Inside the async block we will instantiate the `Resources` struct that loads all the assets. After that we use the

`storage::store()` to save the resources in the Macroquad storage. This will ensure that we can access the resources anywhere in the code.

Using the method `is_done()` on `Coroutine` we can check if the couroutune has finished running or not. We add a loop that runs until `is_done()` returns `true`. While the coroutine is running we use `draw_text()` to display a message on the screen. We also add 1 to 3 periods after the text using the code `".".repeat(((get_time() * 2.) as usize) % 4)`. We also need to use `clear_background()` and `next_frame.await` inside the loop for everything to work properly.

```
    pub async fn load() -> Result<(), macroquad::Error> {
        let resources_loading = start_coroutine(async move {
            let resources = Resources::new().await.unwrap();
            storage::store(resources);
        });

        while !resources_loading.is_done() {
            clear_background(BLACK);
            let text = format!(
                "Loading resources {}",
                ".".repeat(((get_time() * 2.) as usize) % 4)
            );
            draw_text(
                &text,
                screen_width() / 2. - 160.,
                screen_height() / 2.,
                40.,
                WHITE,
            );
            next_frame().await;
        }

        Ok(())
    }
```

Coroutine and Storage API

More information about the Macroquad coroutines (https://docs.rs/macroquad/latest/macroquad/experimental/coroutines/index.html) and storage (https://docs.rs/macroquad/latest/macroquad/experimental/collections/storage/index.html) can be found in the Macroquad documentation.

Loading assets

The call to loading resources needs to be updated to use the new `load()` method instead of using `new()` directly. Since `load()` stores the resources in the Macroquad storage we will use `storage::get::<Resources>()` to retrieve the resources.

```
set_pc_assets_folder("assets");
Resources::load().await?;
let resources = storage::get::<Resources>();
```

Try the game

While the game is loading in a browser, the message “Loading resources...” will be shown on the screen.

> **Challenge: Loading spinner**
>
>
>
> Make a loading spinner by including an image as bytes and draw it using the `rotation` field in `DrawTextureParams` in the `load()` function instead of displaying text.

Release your game

Now that you have made a complete game, you need to release it so that others can play it. In the following chapters are instructions on how to build your game for different platforms.

We'll start by looking at how to build and package the game for the most common desktop platforms: Windows, MacOS, and Linux. After that is a chapter on building the game to run on a web page. We will also look at how to build and package the game for mobile platforms such as Android and iPhone.

Build your game for desktop platforms

Macroquad supports multiple desktop platforms, such as Windows, MacOS, and Linux. It's possible to cross-compile for other platforms than the one you are using. But it might need other tools that are not described in this guide. It's easiest to use a build system that has support for different platforms.

Build for Windows

If you want to build your game to be run on Windows you need to install a Rust build target. Both the MSVC and GNU build targets are supported.

Build using Windows GNU target

Before running the build the for first time you need to install the build target. You will only have to run this command once.

```
rustup target add x86_64-pc-windows-gnu
```

To build the game, use the following command:

```
cargo build --release --target x86_64-pc-windows-gnu
```

The binary file created will be stored in the directory `target/x86_64-pc-windows-gnu/release/`.

Build using Windows MSVC target

Before running the build the for first time you need to install the build target. You will only have to run this command once.

```
rustup target add x86_64-pc-windows-msvc
```

To build the game, use the following command:

```
cargo build --release --target x86_64-pc-windows-msvc
```

The binary file created will be stored in the directory `target/x86_64-pc-windows-msvc/release/`.

Build for Linux

To build your game with Macroquad on Linux you will need a couple of development packages. Below are a few instructions how to install these packages on some common Linux distributions.

Install packages

Ubuntu

These system packages must be installed to build on Ubuntu.

```
apt install pkg-config libx11-dev libxi-dev libgl1-mesa-dev libasound2-dev
```

Fedora

These system packages must be installed to build on Fedora.

```
dnf install libX11-devel libXi-devel mesa-libGL-devel alsa-lib-devel
```

Arch Linux

These system packages must be installed to build on Arch Linux.

```
pacman -S pkg-config libx11 libxi mesa-libgl alsa-lib
```

Build using Linux GNU target

Before running the build for the first time you need to install the build target. You will only have to run this command once.

```
rustup target add x86_64-unknown-linux-gnu
```

To build the game, use the following command:

```
cargo build --release --target x86_64-unknown-linux-gnu
```

The binary file created will be stored in the directory `target/x86_64-unknown-linux-gnu/release/`.

Build using MacOS

To build on MacOS there are two possible targets: `x86_64-apple-darwin` is used for older Intel-based Mac computers, and `aarch64-apple-darwin` build is used for newer Apple Silicon-based Mac computers.

Build using x86-64 Apple Darwin target

Before running the build for the first time you need to install the build target. You will only have to run this command once.

```
rustup target add x86_64-apple-darwin
```

To build the game, use the following command:

```
cargo build --release --target x86_64-apple-darwin
```

The binary file created will be stored in the directory `target/x86_64-apple-darwin/release/`.

Build using aarch64 Apple Darwin target

Before running the build for the first time you need to install the build target. You will only have to run this command once.

```
rustup target add aarch64-apple-darwin
```

To build the game, use the following command:

```
cargo build --release --target aarch64-apple-darwin
```

The binary file created will be stored in the directory `target/aarch64-apple-darwin/release/`.

Package the game

To share your game with others you need to package the game binary file together with all the assets needed to run the game. Here are a couple of examples on how to do this using a terminal.

Windows

```
cp target/x86_64-pc-windows-gnu/release/my-game.exe ./
tar -c -a -f my-game-win.zip my-game.exe assets/*
```

Linux

```
cp target/x86_64-pc-linux-gnu/release/my-game ./
tar -zcf my-game-linux.zip my-game assets/*
```

Mac

```
cp target/aarch64-apple-darwin/release/my-game ./
zip -r my-game-mac.zip my-game assets/*
```

Publish your game on the web

Since you can compile a Macroquad game to WebAssembly it's possible to run the game in a web browser. These are instructions on how to create a web page to run your game. This web page can be published on a web account so that people can play your game directly in the browser without having to download anything.

Install WASM build target

Start by installing the build target for WebAssembly using the command `rustup`.

```
rustup target add wasm32-unknown-unknown
```

Build a WebAssembly binary

Using the WebAssembly target you can build a WASM binary file that can be loaded from a web page.

```
cargo build --release --target wasm32-unknown-unknown
```

The WASM binary file will be placed in the directory `target/wasm32-unknown-unknown/release/` with the extension `.wasm`.

Copy WebAssembly binary

You need to copy the WebAssembly binary to the root of your crate, in the same place where the `assets` directory is placed.

If you have named your crate something else than `my-game`, the name of the binary will have the same name, but with the file extension `.wasm`.

```
cp target/wasm32-unknown-unknown/release/my-game.wasm .
```

Create an HTML page

You will need an HTML page to load the WebAssembly binary. It needs to load a Javascript file from Macroquad which contains code to run the WebAssembly binary and communicate with the browser. You also need to add a canvas element that Macroquad will use to draw the graphics. Remember to change the name of the WebAssembly binary file in the `load()` call from `my-game.wasm` to the name of your game if you have changed it.

Create a file with the name `index.html` in the root of your crate with the following content:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1" />
    <title>My Game</title>
    <style>
        html,
        body,
        canvas {
            margin: 0;
            padding: 0;
            width: 100%;
            height: 100%;
            overflow: hidden;
            position: absolute;
            background: black;
            z-index: 0;
        }
    </style>
</head>
<body>
    <canvas id="glcanvas" tabindex='1'></canvas>
    <!-- Minified and statically hosted version of https://github.com/not-
fl3/macroquad/blob/master/js/mq_js_bundle.js -->
    <script src="mq_js_bundle.js"></script>
    <script>load("my-game.wasm");</script> <!-- Your compiled WASM binary -->
</body>
</html>
```

Test the game in a browser

You should be able to start a web server and open the game in a web browser.

Install a simple web server

This is only to be able to test the game locally before you upload it to a proper web hosting account. To serve your game locally on your computer you can install a simple web server with the following command:

```
cargo install basic-http-server
```

Run the web server

This command will start the web server and print an address where you can reach the web page. Open your web browser and load the URL, this will be something similar to `http://localhost:4000`. The game should now run in your browser instead of as a native application.

```
basic-http-server .
```

Publish your game

If you have access to a web hosting account, you can publish the files there to let other people play your game. You need to upload the HTML file, the WASM file, and the `assets` directory.

```
index.html
my-game.wasm
assets/*
```

> **Publish automatically**
>
> This is a reminder that there are instructions at the end of chapter 1 on how to automatically publish the game on GitHub without using a web account. In that case you need to use the updated `deploy.yml` from chapter 10 – Graphics.

Build for Android phones

Using Macroquad it's possible to build your game to be run on Android phones. We will build an APK file that can be installed on Android phones or added to the Google Play store. We'll describe how to build the game using Docker (https://www.docker.com/get-started/), so you need to have that installed to continue.

Since mobile platforms don't have physical keyboards you will also have to add support for controlling the game using touch controls.

> **Macroquad touch API**
>
> Read about the function `touches()` in the Macroquad documentation (https://docs.rs/macroquad/latest/macroquad/input/index.html) for more information on how touch controls work.

Install the Docker image

Before you build an APK file for Android you need to pull the Docker image `notfl3/cargo-apk`.

```
docker pull notfl3/cargo-apk
```

Build the APK file

Using this command you can build an APK file. It will take quite some time since it will do three full builds, one for each Android target.

```
docker run
  --rm
  -v $(pwd):/root/src
  -w /root/src
  notfl3/cargo-apk cargo quad-apk build --release
```

After this you will have an APK file in the directory `target/android-artifacts/release/apk`.

Configuration

To ensure that Android can find all the assets, you need to add some configuration to the `Cargo.toml` file to define where the assets can be found.

```
[package.metadata.android]
assets = "assets/"
```

> **Detailed Android instructions**
>
> On the Macroquad homepage there are more detailed instructions on how to build for Android (https://macroquad.rs/articles/android/). It has tips on how to speed up the build, how to build manually without Docker, and how to sign the APK file which is needed to upload it to the Google Play Store.

Build for iOS

You can build your Macroquad game to run on iPhone mobile phones and iPads.

> **Adapt to mobile**
>
> You'll notice that the game isn't fully adapted to be run on a mobile platform yet. To start with you can read about the function `touches()` in the Macroquad documentation (https://docs.rs/macroquad/latest/macroquad/input/index.html) or more information about how touch interfaces work.

> **Detailed iOS instructions**
>
> More detailed information on how to build for iOS is available in the article Macroquad on iOS (https://macroquad.rs/articles/ios/) on the Macroquad homepage. There you can find information on how to access logs, building for real devices, and signing your app.

Create a directory

An iOS app is a regular directory with the file extension `.app`.

```
mkdir MyGame.app
```

For our game, the directory structure in the `MyGame.app` directory is the same as when we run the game with `cargo run` from the root of the crate. The binary file and `assets` directory should be placed next to each other. You also need an `Info.plist` file.

Start by adding the `assets`.

```
cp -r assets MyGame.app
```

Build the binary

You need to add the Rust target for iOS. For the simulator you should use Intel binaries and for the real devices you should use ARM binaries. This guide will only cover how to try the game in the simulator. How to try the game on a real device is covered in the Macroquad on iOS (https://macroquad.rs/articles/ios/) article on the Macroquad homepage.

```
rustup target add x86_64-apple-ios
```

After this you can build an executable binary for the iOS Simulator using the following command:

```
cargo build --release --target x86_64-apple-ios
```

Copy the binary file

Copy the executable binary file to the game directory.

```
cp target/x86_64-apple-ios/release/my-game MyGame.app
```

Create Info.plist

Create a text file for the app metadata with the name `Info.plist` in the `MyGame.app` directory with the following content:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
<key>CFBundleExecutable</key>
<string>my-game</string>
<key>CFBundleIdentifier</key>
<string>com.mygame</string>
<key>CFBundleName</key>
<string>mygame</string>
<key>CFBundleVersion</key>
<string>1</string>
<key>CFBundleShortVersionString</key>
<string>1.0</string>
</dict>
</plist>
```

Setup the simulator

For this step you need to have XCode and at least one simulator image installed. You'll find XCode in the App Store app. You can add simulators via the command line or via XCode. In version 15.1 of XCode you can do it via `Settings...` → `Platforms` and then choose between the available iOS versions. There is also a button (`+`) to add more iOS versions.

To add simulators via the command line you first need to run the command `xcrun simctl list` to get a list of all the available simulators. Copy the hex code for the simulator you want and use it as argument to the `xcrun simctl boot` command. You only need to do this the first time you run the simulator.

```
xcrun simctl list
xcrun simctl boot <hex string>
```

Run the simulator

The command we'll use to install and run the game, `xcrun simctl`, chooses a simulator with the argument `booted`. This means that you first need to start a simulator and to make things predictable, you should only run one simulator at a time. This can also be done using the terminal, but the easiest way is to start the Simulator app and then start the simulator you want via `File → Open Simulator`.

To start the simulator using the terminal, use the following command:

```
open /Applications/Xcode.app/Contents/Developer/Applications/Simulator.app/
```

Install the game

You can install the game by dragging the directory `MyGame.app` and dropping it on the running simulator. But since you probably want to reinstall it multiple times it is more efficient to use the terminal with this command:

```
xcrun simctl install booted MyGame.app/
```

Start the game

This can be done using the running simulator or via the terminal. In our `Info.plist` file we specified `CFBundleIdentifier` as `com.mygame`, which we will use to start the game.

```
xcrun simctl launch booted com.mygame
```

The end

You have now developed and published a simple game written in the programming language Rust and the game library Macroquad. While there is still a lot to be done to make it into a complete game, you should now have a solid foundation to improve the game further.

Ideas for improvement

Here are some ideas on how to improve the game to make it more fun to play:

- Add more enemies with different movements and graphics
- Add life to enemies so bigger enemies needs to shot multiple times before they are destroyed
- Allow enemies to shoot bullets or drop bombs themselves
- Make enemies show up in waves instead of just randomly
- Add levels with increasing difficulty
- Upgrades that improve the weapons or add different types of weapons
- Add big boss enemies at the end of levels
- Extra lives
- Add health and display a health bar
- Add an upgrade with a shield around the spaceship
- Store the top ten scores and add a highscore screen
- Use macroquad-tiled (https://github.com/not-fl3/macroquad/tree/master/tiled) to make a level with graphical background
- Add a shop between levels to buy upgrades
- Make the spaceship invulnerable and blinking for a short while after resurrection
- Look at the Macroquad post processing (https://github.com/not-fl3/macroquad/blob/master/examples/post_processing.rs) example on how to add a CRT shader
- Use the font from the graphical menu chapter for all texts in the game
- Show a victory sequence at the end of each level
- Support two simultaneous players

Other resources

These are some other resources for the Macroquad game library.

- Macroquad (https://macroquad.rs) – The official homepage of Macroquad
- Awesome Quads (https://github.com/ozkriff/awesome-quads) – A curated list of Macroquad games and resources
- Quads discord server (https://discord.gg/WfEp6ut) – The official Macroquad community
- Rust game ports (https://github.com/rust-gamedev/rust-game-ports) – Official host of games ported using Rust game libraries

Game showcase

If you have completed this game guide and published your game online, you can have your game showcased on this page. You can open a PR (https://github.com/ollej/macroquad-introduktion/blob/main/macroquad-introduction-book-english/src/wrapup.md) in the GitHub repository and add a link to your game in the list below.

- My first Macroquad game (https://pez.github.io/my-first-macroquad-game/) by Pez (https://www.github.com/pez)
- Implementation with Bevy ECS (https://github.com/ddmills/macroquad-book) by ddmills (https://github.com/ddmills)

Full source code

This is the full source code of the completed game.

```
use macroquad::audio::{Sound, load_sound, play_sound, play_sound_once,
PlaySoundParams};
use macroquad::experimental::animation::{AnimatedSprite, Animation};
use macroquad::experimental::collections::storage;
use macroquad::experimental::coroutines::start_coroutine;
use macroquad::prelude::*;
use macroquad::ui::{hash, root_ui, Skin};
use macroquad_particles::{self as particles, AtlasConfig, Emitter, EmitterConfig};

use std::fs;

const FRAGMENT_SHADER: &str = include_str!("starfield-shader.glsl");

const VERTEX_SHADER: &str = "#version 100
attribute vec3 position;
attribute vec2 texcoord;
attribute vec4 color0;
varying float iTime;

uniform mat4 Model;
uniform mat4 Projection;
uniform vec4 _Time;

void main() {
    gl_Position = Projection * Model * vec4(position, 1);
    iTime = _Time.x;
}
";

struct Shape {
    size: f32,
    speed: f32,
    x: f32,
    y: f32,
    collided: bool,
}

impl Shape {
    fn collides_with(&self, other: &Self) -> bool {
        self.rect().overlaps(&other.rect())
    }

    fn rect(&self) -> Rect {
        Rect {
            x: self.x - self.size / 2.0,
            y: self.y - self.size / 2.0,
            w: self.size,
            h: self.size,
        }
    }
}
```

```
enum GameState {
    MainMenu,
    Playing,
    Paused,
    GameOver,
}

fn particle_explosion() -> particles::EmitterConfig {
    particles::EmitterConfig {
        local_coords: false,
        one_shot: true,
        emitting: true,
        lifetime: 0.6,
        lifetime_randomness: 0.3,
        explosiveness: 0.65,
        initial_direction_spread: 2.0 * std::f32::consts::PI,
        initial_velocity: 400.0,
        initial_velocity_randomness: 0.8,
        size: 16.0,
        size_randomness: 0.3,
        atlas: Some(AtlasConfig::new(5, 1, 0..)),
        ..Default::default()
    }
}

struct Resources {
    ship_texture: Texture2D,
    bullet_texture: Texture2D,
    explosion_texture: Texture2D,
    enemy_small_texture: Texture2D,
    theme_music: Sound,
    sound_explosion: Sound,
    sound_laser: Sound,
    ui_skin: Skin,
}

impl Resources {
    async fn new() -> Result<Resources, macroquad::Error> {
        let ship_texture: Texture2D = load_texture("ship.png").await?;
        ship_texture.set_filter(FilterMode::Nearest);
        let bullet_texture: Texture2D = load_texture("laser-bolts.png").await?;
        bullet_texture.set_filter(FilterMode::Nearest);
        let explosion_texture: Texture2D = load_texture("explosion.png").await?;
        explosion_texture.set_filter(FilterMode::Nearest);
        let enemy_small_texture: Texture2D = load_texture("enemy-small.png").await?;
        enemy_small_texture.set_filter(FilterMode::Nearest);
        build_textures_atlas();

        let theme_music = load_sound("8bit-spaceshooter.ogg").await?;
        let sound_explosion = load_sound("explosion.wav").await?;
        let sound_laser = load_sound("laser.wav").await?;

        let window_background = load_image("window_background.png").await?;
        let button_background = load_image("button_background.png").await?;
        let button_clicked_background =
load_image("button_clicked_background.png").await?;
        let font = load_file("atari_games.ttf").await?;
```

```
    let window_style = root_ui()
        .style_builder()
        .background(window_background.clone())
        .background_margin(RectOffset::new(32.0, 76.0, 44.0, 20.0))
        .margin(RectOffset::new(0.0, -40.0, 0.0, 0.0))
        .build();
    let button_style = root_ui()
        .style_builder()
        .background(button_background.clone())
        .background_clicked(button_clicked_background.clone())
        .background_margin(RectOffset::new(16.0, 16.0, 16.0, 16.0))
        .margin(RectOffset::new(16.0, 0.0, -8.0, -8.0))
        .font(&font)?
        .text_color(WHITE)
        .font_size(64)
        .build();
    let label_style = root_ui()
        .style_builder()
        .font(&font)?
        .text_color(WHITE)
        .font_size(28)
        .build();
    let ui_skin = Skin {
        window_style,
        button_style,
        label_style,
        ..root_ui().default_skin()
    };

    Ok(Resources {
        ship_texture,
        bullet_texture,
        explosion_texture,
        enemy_small_texture,
        theme_music,
        sound_explosion,
        sound_laser,
        ui_skin,
    })
}

pub async fn load() -> Result<(), macroquad::Error> {
    let resources_loading = start_coroutine(async move {
        let resources = Resources::new().await.unwrap();
        storage::store(resources);
    });

    while !resources_loading.is_done() {
        clear_background(BLACK);
        let text = format!(
            "Loading resources {}",
            ".".repeat(((get_time() * 2.) as usize) % 4)
        );
        draw_text(
            &text,
            screen_width() / 2. - 160.,
            screen_height() / 2.,
            40.,
            WHITE,
```

```
            );
            next_frame().await;
        }

        Ok(())
    }
}

#[macroquad::main("My game")]
async fn main() -> Result<(), macroquad::Error> {
    const MOVEMENT_SPEED: f32 = 200.0;

    rand::srand(miniquad::date::now() as u64);
    let mut squares = vec![];
    let mut bullets: Vec<Shape> = vec![];
    let mut circle = Shape {
        size: 32.0,
        speed: MOVEMENT_SPEED,
        x: screen_width() / 2.0,
        y: screen_height() / 2.0,
        collided: false,
    };
    let mut score: u32 = 0;
    let mut high_score: u32 = fs::read_to_string("highscore.dat")
        .map_or(Ok(0), |i| i.parse::<u32>())
        .unwrap_or(0);
    let mut game_state = GameState::MainMenu;

    let mut direction_modifier: f32 = 0.0;
    let render_target = render_target(320, 150);
    render_target.texture.set_filter(FilterMode::Nearest);
    let material = load_material(
        ShaderSource::Glsl {
            vertex: VERTEX_SHADER,
            fragment: FRAGMENT_SHADER,
        },
        MaterialParams {
            uniforms: vec![
                UniformDesc::new("iResolution", UniformType::Float2),
                UniformDesc::new("direction_modifier", UniformType::Float1),
            ],
            ..Default::default()
        },
    )?;

    let mut explosions: Vec<(Emitter, Vec2)> = vec![];

    set_pc_assets_folder("assets");
    Resources::load().await?;
    let resources = storage::get::<Resources>();

    let mut bullet_sprite = AnimatedSprite::new(
        16,
        16,
        &[
            Animation {
                name: "bullet".to_string(),
                row: 0,
                frames: 2,
```

```
            fps: 12,
        },
        Animation {
            name: "bolt".to_string(),
            row: 1,
            frames: 2,
            fps: 12,
        },
    ],
    true,
);
bullet_sprite.set_animation(1);
let mut ship_sprite = AnimatedSprite::new(
    16,
    24,
    &[
        Animation {
            name: "idle".to_string(),
            row: 0,
            frames: 2,
            fps: 12,
        },
        Animation {
            name: "left".to_string(),
            row: 2,
            frames: 2,
            fps: 12,
        },
        Animation {
            name: "right".to_string(),
            row: 4,
            frames: 2,
            fps: 12,
        },
    ],
    true,
);
let mut enemy_small_sprite = AnimatedSprite::new(
    17,
    16,
    &[Animation {
        name: "enemy_small".to_string(),
        row: 0,
        frames: 2,
        fps: 12,
    }],
    true,
);

play_sound(
    &resources.theme_music,
    PlaySoundParams {
        looped: true,
        volume: 1.,
    },
);

root_ui().push_skin(&resources.ui_skin);
let window_size = vec2(370.0, 320.0);
```

```
loop {
    clear_background(BLACK);

    material.set_uniform("iResolution", (screen_width(), screen_height()));
    material.set_uniform("direction_modifier", direction_modifier);
    gl_use_material(&material);
    draw_texture_ex(
        &render_target.texture,
        0.,
        0.,
        WHITE,
        DrawTextureParams {
            dest_size: Some(vec2(screen_width(), screen_height())),
            ..Default::default()
        },
    );
    gl_use_default_material();

    match game_state {
        GameState::MainMenu => {
            root_ui().window(
                hash!(),
                vec2(
                    screen_width() / 2.0 - window_size.x / 2.0,
                    screen_height() / 2.0 - window_size.y / 2.0,
                ),
                window_size,
                |ui| {
                    ui.label(vec2(80.0, -34.0), "Main Menu");
                    if ui.button(vec2(65.0, 25.0), "Play") {
                        squares.clear();
                        bullets.clear();
                        explosions.clear();
                        circle.x = screen_width() / 2.0;
                        circle.y = screen_height() / 2.0;
                        score = 0;
                        game_state = GameState::Playing;
                    }
                    if ui.button(vec2(65.0, 125.0), "Quit") {
                        std::process::exit(0);
                    }
                },
            );
        }
        GameState::Playing => {
            let delta_time = get_frame_time();
            ship_sprite.set_animation(0);
            if is_key_down(KeyCode::Right) {
                circle.x += MOVEMENT_SPEED * delta_time;
                direction_modifier += 0.05 * delta_time;
                ship_sprite.set_animation(2);
            }
            if is_key_down(KeyCode::Left) {
                circle.x -= MOVEMENT_SPEED * delta_time;
                direction_modifier -= 0.05 * delta_time;
                ship_sprite.set_animation(1);
            }
            if is_key_down(KeyCode::Down) {
```

```
    circle.y += MOVEMENT_SPEED * delta_time;
}
if is_key_down(KeyCode::Up) {
    circle.y -= MOVEMENT_SPEED * delta_time;
}
if is_key_pressed(KeyCode::Space) {
    bullets.push(Shape {
        x: circle.x,
        y: circle.y - 24.0,
        speed: circle.speed * 2.0,
        size: 32.0,
        collided: false,
    });
    play_sound_once(&resources.sound_laser);
}
if is_key_pressed(KeyCode::Escape) {
    game_state = GameState::Paused;
}

// Clamp X and Y to be within the screen
circle.x = clamp(circle.x, 0.0, screen_width());
circle.y = clamp(circle.y, 0.0, screen_height());

// Generate a new square
if rand::gen_range(0, 99) >= 95 {
    let size = rand::gen_range(16.0, 64.0);
    squares.push(Shape {
        size,
        speed: rand::gen_range(50.0, 150.0),
        x: rand::gen_range(size / 2.0, screen_width() - size / 2.0),
        y: -size,
        collided: false,
    });
}

// Movement
for square in &mut squares {
    square.y += square.speed * delta_time;
}
for bullet in &mut bullets {
    bullet.y -= bullet.speed * delta_time;
}

ship_sprite.update();
bullet_sprite.update();
enemy_small_sprite.update();

// Remove shapes outside of screen
squares.retain(|square| square.y < screen_height() + square.size);
bullets.retain(|bullet| bullet.y > 0.0 - bullet.size / 2.0);

// Remove collided shapes
squares.retain(|square| !square.collided);
bullets.retain(|bullet| !bullet.collided);

// Remove old explosions
explosions.retain(|(explosion, _)| explosion.config.emitting);

// Check for collisions
```

```
                if squares.iter().any(|square| circle.collides_with(square)) {
                    if score == high_score {
                        fs::write("highscore.dat", high_score.to_string()).ok();
                    }
                    game_state = GameState::GameOver;
                }
                for square in squares.iter_mut() {
                    for bullet in bullets.iter_mut() {
                        if bullet.collides_with(square) {
                            bullet.collided = true;
                            square.collided = true;
                            score += square.size.round() as u32;
                            high_score = high_score.max(score);
                            explosions.push((
                                Emitter::new(EmitterConfig {
                                    amount: square.size.round() as u32 * 4,
                                    texture:
Some(resources.explosion_texture.clone()),
                                    ..particle_explosion()
                                }),
                                vec2(square.x, square.y),
                            ));
                            play_sound_once(&resources.sound_explosion);
                        }
                    }
                }

                // Draw everything
                let bullet_frame = bullet_sprite.frame();
                for bullet in &bullets {
                    draw_texture_ex(
                        &resources.bullet_texture,
                        bullet.x - bullet.size / 2.0,
                        bullet.y - bullet.size / 2.0,
                        WHITE,
                        DrawTextureParams {
                            dest_size: Some(vec2(bullet.size, bullet.size)),
                            source: Some(bullet_frame.source_rect),
                            ..Default::default()
                        },
                    );
                }
                let ship_frame = ship_sprite.frame();
                draw_texture_ex(
                    &resources.ship_texture,
                    circle.x - ship_frame.dest_size.x,
                    circle.y - ship_frame.dest_size.y,
                    WHITE,
                    DrawTextureParams {
                        dest_size: Some(ship_frame.dest_size * 2.0),
                        source: Some(ship_frame.source_rect),
                        ..Default::default()
                    },
                );
                let enemy_frame = enemy_small_sprite.frame();
                for square in &squares {
                    draw_texture_ex(
                        &resources.enemy_small_texture,
                        square.x - square.size / 2.0,
```

```
                        square.y - square.size / 2.0,
                        WHITE,
                        DrawTextureParams {
                            dest_size: Some(vec2(square.size, square.size)),
                            source: Some(enemy_frame.source_rect),
                            ..Default::default()
                        },
                    );
                }
                for (explosion, coords) in explosions.iter_mut() {
                    explosion.draw(*coords);
                }
                draw_text(
                    format!("Score: {}", score).as_str(),
                    10.0,
                    35.0,
                    25.0,
                    WHITE,
                );
                let highscore_text = format!("High score: {}", high_score);
                let text_dimensions = measure_text(highscore_text.as_str(), None, 25,
1.0);
                draw_text(
                    highscore_text.as_str(),
                    screen_width() - text_dimensions.width - 10.0,
                    35.0,
                    25.0,
                    WHITE,
                );
            }
            GameState::Paused => {
                if is_key_pressed(KeyCode::Escape) {
                    game_state = GameState::Playing;
                }
                let text = "Paused";
                let text_dimensions = measure_text(text, None, 50, 1.0);
                draw_text(
                    text,
                    screen_width() / 2.0 - text_dimensions.width / 2.0,
                    screen_height() / 2.0,
                    50.0,
                    WHITE,
                );
            }
            GameState::GameOver => {
                if is_key_pressed(KeyCode::Space) {
                    game_state = GameState::MainMenu;
                }
                let text = "GAME OVER!";
                let text_dimensions = measure_text(text, None, 50, 1.0);
                draw_text(
                    text,
                    screen_width() / 2.0 - text_dimensions.width / 2.0,
                    screen_height() / 2.0,
                    50.0,
                    RED,
                );
            }
        }
```

```
            next_frame().await
        }
    }
```

Credits

Ferris the Gamer

The image of Ferris holding a game controller is adapted from the “Ferris the Rustacean” public domain image created by Karen Rustad Tölva (https://www.rustacean.net/). The game controller is drawn by Clovis_Cheminot from Pixabay (https://pixabay.com/vectors/controller-video-game-flat-x-box-1486898/) and released under the Pixabay Content License.

Ferris the Teacher

The image Ferris the Teacher (https://www.behance.net/gallery/89117181/Ferris-the-professional) is made by Esther Arzola (https://www.behance.net/estherarzola) licensed under CC BY 4.0 (https://creativecommons.org/licenses/by/4.0/deed.en).

Starfield shader

The starfield shader is created by The Art of Code and taken from the video Shader Coding: Making a starfield (https://youtu.be/rvDo9LvfoVE). License: Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Unported License.

Asset credits

Sprites

Space Ship Shooter Pixel Art Assets
Author: ansimuz
License: CC0 Public Domain
https://opengameart.org/content/space-ship-shooter-pixel-art-assets
(https://opengameart.org/content/space-ship-shooter-pixel-art-assets)

Theme music

8-bit space shooter music
Author: HydroGene
License: CC0 Public Domain
https://opengameart.org/content/8-bit-epic-space-shooter-music
(https://opengameart.org/content/8-bit-epic-space-shooter-music)

Laser and explosion sounds

Sci-fi sounds
Author: Kenney.nl
License: CC0 Public Domain
https://opengameart.org/content/sci-fi-sounds (https://opengameart.org/content/sci-fi-sounds)

UI

Sci-fi User Interface Elements
Author: Buch
License: CC0 Public Domain
sci-fi-ui.psd
https://opengameart.org/content/sci-fi-user-interface-elements
(https://opengameart.org/content/sci-fi-user-interface-elements)

Font

AtariGames
Author: Kieran
License: Public Domain
https://nimblebeastscollective.itch.io/nb-pixel-font-bundle (https://nimblebeastscollective.itch.io/nb-pixel-font-bundle)

Glossary

This is a list of terms and abbreviations used in this guide.

Word	Definition
APK	Android package is a file format to distribute mobile apps for the Android OS
camera	A virtual camera used to project a 3D world on a 2D screen
cargo	The package manager for Rust
Docker	A tool to run applications in isolated containers
enum	A Rust feature, to enumerate its possible variants
GitHub	A developer platform to create, store, manage, and share code
GLSL	OpenGL Shading Language executed on the GPU
GNU	A mass collaboration project that creates free software
GPU	Graphics processing unit, an electronic circuit specialized to accelerate computer graphics
Javascript	A programming language that can run in web browsers
Macroquad	A game library to write games with Rust
Miniquad	A small Rust graphics library used by Macroquad
MP3	A sound file format
MSVC	Microsoft Visual C++ compiler
Ogg Vorbis	A sound file format
OGG	The filename extension for Ogg Vorbis sound files
particle system	A computer graphics technique using many small sprites to simulate visual effects
PNG	An image file format
PR	Pull requests are a way to propose changes to code in a distributed version control system
Rust	A programming language
shader	A small program that runs on the GPU
sprite	A fixed-size 2D image used as a part of a larger scene
Struct	A Rust custom data type, used to structure related values
terminal	A program used to run text commands
texture	An image stored in the GPU memory
Vec2	A 2-dimensional vector
Vector	A Rust collection that can store a variable number of values
Vsync	Vertical sync ensures the monitor displays every frame the GPU renders
WASM	Abbreviation of WebAssembly
WebAssembly	A binary instruction format that can run in web browsers
XCode	An IDE by Apple to develop software for their platforms